Title	UV-Diagram: A Voronoi Diagram for Uncertain Spatial Databases
Author(s)	Xie, X; Cheng, CK; Yiu, ML; Sun, L; Chen, J
Citation	Very Large Databases Journal (VLDBJ), 2013, v. 22 n. 3, p. 319-344
Issued Date	2013
URL	http://hdl.handle.net/10722/165825
Rights	Creative Commons: Attribution 3.0 Hong Kong License

The VLDB Journal (2013) 22:319–344
DOI 10.1007/s00778-012-0290-x

REGULAR PAPER

UV-diagram: a voronoi diagram for uncertain spatial databases

Xike Xie · Reynold Cheng · Man Lung Yiu · Liwen Sun · Jinchuan Chen

Received: 21 May 2011 / Revised: 24 May 2012 / Accepted: 7 August 2012 / Published online: 7 September 2012

Abstract The Voronoi diagram is an important technique for answering nearest-neighbor queries for spatial databases. We study how the Voronoi diagram can be used for uncertain spatial data, which are inherent in scientific and business applications. Specifically, we propose the *Uncertain-Voronoi diagram* (or *UV-diagram*), which divides the data space into disjoint "UV-partitions". Each UV-partition P is associated with a set S of objects, such that any point q located in P has the set S as its nearest neighbor with nonzero probabilities. The UV-diagram enables queries that return objects with nonzero chances of being the nearest neighbor (NN) of a given point q. It supports "continuous nearest-neighbor search", which refreshes the set of NN objects of q, as the position of q changes. It also allows the analysis of nearest-neighbor information, for example, to find out the number of objects that are the nearest neighbors of any point in a given area. A UV-diagram requires exponential construction and storage costs. To tackle these problems, we devise an alternative representation of a UV-diagram, by using a set of *UV-cells*. A UV-cell of an object o is the extent e for which o can be the nearest neighbor of any point $q \in e$. We study how to speed up the derivation of UV-cells by considering its nearby objects. We also use the UV-cells to design the *UV-index*, which supports different queries, and can be constructed in polynomial time. We have performed extensive experiments on both real and synthetic data to validate the efficiency of our approaches.

Keywords Voronoi diagram · Uncertain data · Nearest-neighbor query

X. Xie
Department of Computer Science, Aalborg University, Aalborg, Denmark
e-mail: xkxie@cs.aau.dk

R. Cheng (✉)
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
e-mail: ckcheng@cs.hku.hk

M. L. Yiu
Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong
e-mail: csmlyiu@comp.polyu.edu.hk

L. Sun
University of California, Berkeley, CA, USA
e-mail: liwen@cs.berkeley.edu

J. Chen
Key Lab for Data Engineering and Knowledge Engineering, MOE. Renmin University of China, Oakland, CA, USA
e-mail: jcchen@ruc.edu.cn

1 Introduction

The Voronoi diagram, primarily designed for evaluating nearest-neighbor queries over two-dimensional spatial point [33], has raised plenty of research interest. This technique has been extended to handle different related problems, including database services in wireless broadcast environments [44,45]; high-dimensional query evaluation [7]; continuous location-based services [4,32,43]; and virus spread analysis among mobile devices [41]. Conceptually, the Voronoi diagram partitions the data space into disjoint "Voronoi cells", so that all points in the same Voronoi cell have the same nearest neighbor. The task of finding the nearest neighbor of a query point is then reduced to a point query. Figure 1a illustrates a Voronoi diagram of seven points. Since the query point q is located in the Voronoi cell of O_2, O_2 is the nearest neighbor of q.

Is it possible to use the Voronoi diagram to perform nearest-neighbor search on objects whose values are impre-

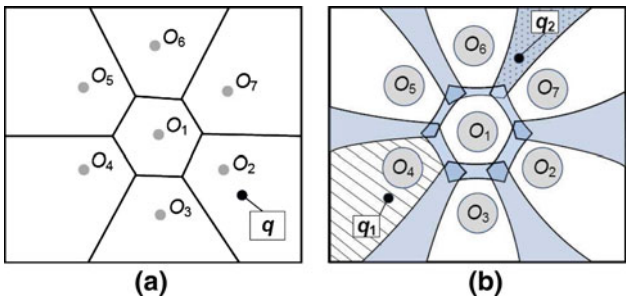


Fig. 1 **a** Voronoi diagram. **b** UV-diagram

cise? Data values can be uncertain for a variety of reasons. Consider a satellite image, which depicts geographical objects like airports, vehicles, and people. Using machine learning and human effort (e.g., community-based systems like Wikimapia), the location of each object on the image can be obtained. Due to the noisy transmission of satellite data, the quality of these images can be affected, and we may not be able to obtain very accurate locations. Moreover, if this location information is released to the public (e.g, for research purposes), it may need to be preprocessed for privacy purposes. In fact, recent proposals like [1,2] have suggested to represent a user's position as a larger region, in order to lower the likelihood that a user is identified at a particular site. Uncertainty is also inherent in biological data management. For example, microscopy images have been actively used to analyze the thickness of neuron layers in the retina, as well as the extent of the area of a cell. Due to factors like image resolution and measurement accuracy, it is hard to obtain exact values of the objects of interest [28,29]. For this kind of data, techniques for evaluating range queries, nearest-neighbor queries, and joins have been developed. These queries return answers with probabilistic guarantees, which reflect the confidence of answers due to data uncertainty. For these applications, tools that resemble the Voronoi diagram can be potentially useful. Specifically, we would like to examine space-partitioning techniques for performing a *Probabilistic Nearest-Neighbor Query* (PNN). Given a query point q, a PNN returns the IDs of objects with nonzero probabilities for being the closest to q, as well as their probabilities. In the sequel, we denote the objects returned by the PNN as *answer objects*, and their probability values as *qualification probabilities*.

An uncertainty model that has been commonly used is to assume that an object O_i has an "uncertainty region" and a probability distribution function (pdf). This means that the precise position of O_i can only be located inside the (closed) region, with a pdf that describes the distribution of the object's position within the region. The uncertainty region can have any shape, and the pdf is arbitrary (e.g., it can be a uniform distribution, Gaussian, or a histogram). Here, we assume that O_i has a two-dimensional circular uncertainty region. We will also explain how our solution can be extended to handle non-circular-shaped regions. We examine how the Voronoi diagram should be defined to support PNN execution. Specifically, we propose the *Uncertain-Voronoi* diagram (or *UV-diagram*), where the nearest-neighbor information of every point in the data space is recorded, based on the uncertain objects involved. The UV-diagram provides a basis for studying solutions that used the Voronoi diagram for point data. It could be interesting, for instance, to extend the solution of [44] to support uncertain data in broadcasting services. Figure 1b illustrates an example of the UV-diagram of seven uncertain objects, where the space is divided into disjoint regions called *UV-partitions*. Each UV-partition P is associated with a set S of one or more objects. For any point q located inside P, S is the set of answer objects of q (i.e., each object in S has a nonzero probability for being the nearest neighbor of q). The highlighted regions contain points that have two or more nearest-neighbor objects. As an example, since q_1 is inside the dashed region, O_4 has a nonzero probability for being the nearest neighbor of q_1; on the other hand, q_2 is located inside the dotted region, and O_6 and O_7 are the answer objects for the PNN with q_2 as the query point. Observe that the Voronoi diagram, which indexes on spatial points, is a special case of the UV-diagram, since a point can be viewed as an uncertainty region with a zero radius. Figure 1 compares the two diagrams.

The Voronoi diagram can also be used in other applications. For example, a *continuous nearest-neighbor query*, which constantly returns the nearest neighbor (e.g., gas station) of a moving point q (e.g., a vehicle), is a typical operation in location-based services [32,43]. The Voronoi diagram supports this query; particularly, the Voronoi cell that contains the current location of q can be easily retrieved. We will illustrate how to use the UV-diagram to track the possible nearest neighbors of a moving point. Another use of the Voronoi diagram is to perform data analysis or observe interesting patterns of nearest-neighbor information. In [41], the Voronoi diagram is used to investigate the spreading pattern of bluetooth viruses among mobile users. We can also use UV-diagram to provide valuable information about these "nearest-neighbor patterns". In Fig. 1b, if the dashed region is large, it indicates that O_4 has high chance to be placed in different clusters (assuming that a nearest-neighbor clustering algorithm is used). Another interesting query is as follows: given a region R, display all UV-partitions that intersect with R, as well as the density of objects that can be the nearest neighbor in each UV-partition. Hence, a UV-diagram allows a user to visualize patterns about the nearest-neighbor information.

Challenges of constructing UV-diagram. Although the UV-diagram is useful, developing a UV-diagram is not simple. Notice that the UV-partitions are produced based on uncertainty regions, which may not be points. Unfortunately,

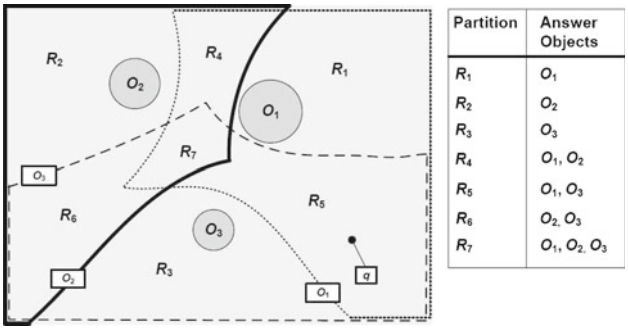


Fig. 2 A UV-diagram for 3 uncertain objects

efficient computational geometry methods for generating the Voronoi diagram (e.g., line-sweeping [19]) cannot be readily used for creating a UV-diagram, since these methods are primarily designed for spatial points, rather than uncertainty regions. Figure 2 depicts the space partition based on three uncertainty regions represented as circles. Each UV-partition (named R_i, where $i = 1, \ldots, 7$) is irregular in shape and contains different answer objects, listed on the side of the figure. In general, given a set of uncertain regions, an exponential number of UV-partitions can be created. For example, Fig. 2 shows that for three objects, there are seven UV-partitions, each of which contains one of $2^3 - 1 = 7$ combinations of the three objects. Moreover, the number of edges of each UV-partition can also be exponentially large! This makes it computationally infeasible to generate and store these partitions. It is also difficult to find out which of these irregular UV-partitions contain a given query point. Indeed, our experimental results show that a brute-force approach of computing and indexing UV-partitions over 40,000 objects requires about 60 h. Therefore, a scalable method for constructing a UV-diagram is highly desirable.

Our solutions. Instead of computing UV-partitions, we have developed an alternative interpretation of the UV-diagram. For every object O_i, we consider the extent a_i such that O_i can be the nearest neighbor of any point selected from a_i. We call this extent the *UV-cell* of O_i. We examine some basic properties of a UV-cell (e.g., its size and number of edges). We show how to represent a UV-cell as a set of objects, and develop novel methods to find this object set efficiently. For example, our *batch-construction* algorithm allows the UV-cells of objects that are physically close to each other to be swiftly obtained. We propose a polynomial time method for constructing an index for the UV-partitions, called the UV-index. We adopt an adaptive-grid indexing scheme, which has the advantage of adapting to different distributions of uncertain objects' positions. Our experimental results show that on both synthetic and real dataset, this index can be constructed in a much shorter time. We also explain how to use the UV-index to support different applications (e.g., PNN and nearest-neighbor pattern queries).

To summarize, our contributions are the following:

- Study the UV-diagram and its basic properties;
- Propose efficient algorithms for obtaining a UV-cell;
- Design the UV-index;
- Use the UV-index to support different queries; and
- Conduct experiments on real and synthetic datasets.

The rest of the paper is as follows. Section 2 summarizes the related work. In Sect. 3, we present basic concepts of the UV-diagram. In Sect. 4, we study the UV-cell and its essential properties. We then explain how to represent UV-cell efficiently in Sect. 5. An adaptive index based on the UV-diagram is presented in Sect. 6. We present experimental results in Sect. 7. Section 8 concludes the paper.

2 Related work

Data uncertainty management. Recently, researchers have proposed to consider uncertainty as a "first-class citizen" in a DBMS [13,14,18,39]. Two models can be used to represent uncertain data: tuple- and attribute- uncertainty. For tuple uncertainty, each database tuple has a probability of being correct [18]. Here, we assume attribute uncertainty, which represents an attribute as a range of possible values and a probability distribution function (pdf) bounded in the range [39]. Common queries for attribute uncertainty include range queries [16], k-nearest-neighbors [28], skylines [25,36], and top-k queries [20].

A few works have been proposed to evaluate PNN queries over attribute uncertainty. In [15], numerical integration techniques have been presented. *Probabilistic verifiers*, described in [13], can generate answer objects' probability bounds without performing expensive integration operations. Another way to compute answer probabilities is based on sampling [24]. In this paper, we focus on the efficient retrieval of answer objects.

To our understanding, the only indexing method available for nearest-neighbor search over uncertain data is to use an index like the R-tree and the grid. The R-tree is a disk-based structure that uses the minimum bounding rectangles (MBRs in short) to cluster the uncertainty regions of the objects, and organizes MBRs in a hierarchical manner [6]. To evaluate PNN using the R-tree, a branch-and-prune strategy has been proposed in [15], where MBRs that may contain answer objects are traversed. However, this involves a lot of I/O cost in reading index nodes and leaf pages [13,15]. Similar issues also occur with grids [31]. On the other hand, retrieving answer objects from the UV-diagram is essentially a point query search: given a point q, find the objects associated with the UV-partition that contains q. Hence, a UV-diagram can support more efficient PNN search.

While it is not clear how an R-tree or grid over uncertain objects can provide pattern analysis of nearest-neighbor information (e.g., displaying the extent of a UV-partition), we will show how to use the UV-diagram to provide this information.

Other types of nearest-neighbor queries, like the "group nearest-neighbors" [26], "reverse-nearest-neighbors" [10,27], "uncertain queries" [8], and "continuous nearest-neighbor queries" [12] have also been proposed. In these works, the R-tree was used to support object retrieval. It is interesting to study how the UV-diagram can be used to support the execution of these queries. In this paper, we study how to use the UV-diagram to support the execution of continuous nearest-neighbor queries.

The Voronoi diagram is an important technique for answering nearest-neighbor queries over spatial points [33]. It has been extended to support other applications (e.g., [7,32,43–45]). It also facilitates the analysis of spreading patterns of mobile viruses [41]. In [9], the k-th order Voronoi diagram is used to evaluate a k-NN query. In [38], an index called VoR-Tree is designed to merge Voronoi diagrams into R-tree in order to answer various nearest-neighbor queries. The Voronoi diagram has also been defined for boundaries of circular objects in [23]. However, these objects are *not* uncertain, and the method of [23] cannot be used to answer PNN queries.

Few works have studied the application of the Voronoi diagram on uncertain data. [8] Consider the "uncertain" nearest neighbor query (UNN) over spatial points. Different from PNN, the query is an uncertain region, not a query point. To evaluate a UNN, the authors propose to use a Voronoi diagram over 2D points. The portions of the Voronoi cells that overlap with the query's uncertainty region are then used to compute answer probabilities. [22] Consider the clustering of uncertain attribute data, where a Voronoi diagram is constructed for centroid points. Notice that [8,22] do not construct a Voronoi diagram for uncertain data. On the other hand, the UV-diagram is a Voronoi diagram tailored for attribute uncertainty.

In [21,37], the Voronoi diagram was modified to identify an imprecise object that is surely the nearest object of a query point q. However, the UV-diagram returns object(s) that *may* have chance to be the nearest neighbor of q, and can be used to answer probabilistic nearest-neighbor queries. We also study a database index for the UV-diagram, which has not been examined in these two works.

This paper is an extension of [17]. Here, we improve the performance of UV-index construction by proposing *batch pruning*, which reduces the workload of generating UV-cells for a set of nearby objects. We provide a more detailed study of the basic properties of a UV-cell (e.g., its size and number of edges). We also examine how the UV-index can be used to answer PNN queries for a moving query point. We conduct new experiments to validate the effectiveness of these approaches.

3 The UV-diagram

We now present the basic notions of the UV-diagram.

We introduce the UV-cell, an alternative presentation of the UV-diagram, in Sect. 3.1. We then study some applications of the UV-diagram, in Sect. 3.2.

3.1 The UV-cell

As discussed before, the UV-diagram can be expensive to construct. We hence propose an alternative representation of the UV-diagram, by using *UV-cells*. We will later explain how the UV-cells facilitate efficient construction of the UV-diagram. Now, let $O_1, O_2, \ldots, O_n$ be the IDs of a set O of uncertain objects, and D be a two-dimensional space that contains these objects. For simplicity, we assume that D is a square. The UV-cell is then defined as follows:

Definition 1 A **UV-cell** of O_i, denoted by U_i, is a region in D such that O_i has a nonzero probability to be the nearest neighbor (NN) of a point q, where $q \in U_i$.

Figure 2 illustrates the UV-cells for O_1, O_2, and O_3. The boundary of each UV-cell is labeled with the ID of the object. For example, the UV-cell of O_2 is a region enclosed by solid-line segments.

The UV-cell can be used to recover the UV-partitions (i.e., disjoint regions of a UV-diagram). In fact, a UV-partition that contains q is the *intersection* of all UV-cells that contain q. This is because the objects associated with these UV-cells have nonzero qualification probabilities for q. For instance, in Fig. 2, the UV-cells of both O_1 and O_3 intersect at partitions R_5 and R_7. This means that when q is located at any of these partitions, both O_1 and O_3 are the answer objects. Since R_7 is intersected by O_2's UV-cell, O_2 is also associated with R_7. Therefore, a UV-diagram is the union of all objects' UV-cells. Besides, the UV-cells of all objects can be used to output which object(s) is/are the nearest neighbor of q with nonzero probabilities.

Table 1 shows the symbols used in this paper. Notice that if there is at least one uncertain object in domain D, any point in D must be covered by at least one UV-cell. In particular, if O_i is the only object in domain D, then its UV-cell is exactly D.

3.2 Applications of the UV-diagram

The UV-diagram supports a number of applications. Let us now explain how to use the UV-diagram to handle the following queries:

Table 1 Notations and meanings

Notation	Meaning
Objects and query	
D	Domain space (a square)
O	A set of uncertain objects ($O_1, O_2, \ldots, O_n$)
$MBC(O_i)$	Minimum bounding circle of object O
(c_i, r_i)	Center and radius of O_i's uncertainty region
q	Query point of a PNN
ρ	Density of objects in D
UV-diagram	
$\odot(c, r)$	A circle centered at c with radius r
$dist(q, c_i)$	Euclidean distance between q and c_i
$dist_{min}(q, O_i)$	Minimum distance of O_i from q
$dist_{max}(q, O_i)$	Maximum distance of O_i from q
U_i	UV-cell of O_i
P_i	Possible region of O_i
$E_i(j)$	UV-edge of O_i w.r.t. O_j
$X_i(j)$ ($\overline{X_i(j)}$)	Outside (inside) region of O_i w.r.t. O_j
F_i	r-Objects of O_i, where $F_i \subseteq O$
C_i	cr-Objects of O_i, where $C_i \subseteq O$
M	Maximum no. of non-leaf nodes
s	Estimated size of a UV-cell
T_θ	Split threshold

1. The Probabilistic Nearest-neighbor (PNN) Query. This query has been mentioned in Sect. 1. To evaluate a PNN for a given point q, we can find out the UV-partition that contains q. The set A of objects associated with this partition are those that can be the nearest neighbor of q. Notice that the UV-partitions can be obtained by finding the union of all the UV-cells. For each object $O_i \in A$, the probability that O_i is the closest to q can be efficiently evaluated by using solutions in [13,15,24].

2. The Continuous PNN Search (CPNN), an extension of the PNN, is a query that resides in the processing server for an extensive period of time. Different from PNN, the position of a query point q changes with time [12]. The objective of the CPNN is to refresh the PNN answer, when the value of q changes. This query can be used in transportation services, where q can be a moving vehicle or person, and the data can be the geographical objects retrieved from satellite images. Assuming that q reports its position to the server periodically, the UV-diagram can conveniently support CPNN. Suppose that the server receives a new position of q, say, q_1. A simple solution is to issue a new PNN for q_1. However, if q_1 is located in the same UV-partition that contains the old position of q, then it suffices to use the objects associated with that UV-partition to compute the query answer for q_1. The cost of retrieving the UV-partition that contains q_1 is thus saved.

3. The UV-partition Query. The UV-diagram can also be used to retrieve the distribution and pattern information about nearest neighbors, which can be useful for analysis purposes (e.g., [41]). One such "pattern-analysis" operation is the *UV-partition query*. Given a region R, this query retrieves all UV-partitions inside R and the "density" of each partition R_j (which is equal to the number of objects associated with R_j, divided by the area of R_j). This allows a user to examine the density distribution of the nearest neighbors in his/her interested area R.

4. The UV-cell Query. This is another pattern analysis operation. Given an object O_i, it returns the extent and the area of O_i's UV-cell. The query user can then obtain the area of the region where O_i may be the nearest neighbor. This area can reflect the "influence" of O_i (in terms of the nearest neighbor information). The shape of the UV-cell can also be displayed on the user's computer screen for further analysis.

Since the UV-diagram is expensive to construct, in Sect. 6, we revisit how the above queries can be implemented by the UV-index, which is an approximate version of the UV-diagram. We next address the UV-cell in detail.

4 More about UV-cells

We now investigate the UV-cell, which is important for constructing the UV-index, in more details. We first present a simple method for constructing a UV-cell in Sect. 4.1. We then examine the shape of a UV-cell in Sect. 4.2. The number of edges of a UV-cell, and its size, is studied in Sects. 4.3 and 4.4, respectively.

4.1 Constructing a UV-cell

Let us first address the relationship between a query point and UV-cells. Let p be a point in D, and let $dist_{min}(O_i, p)$ and $dist_{max}(O_i, p)$ be the minimum and the maximum distances of object O_i from p, respectively. Figure 3 illustrates two uncertain objects, O_i and O_j. For any point p on the solid line shown, we require the following property to hold:

$$dist_{min}(O_i, p) = dist_{max}(O_j, p) \quad (1)$$

We call this solid line the "UV-edge of O_i with respect to O_j", denoted by $E_i(j)$. A special property of this edge is that any point p at the region on the side of $E_i(j)$ closer to O_j has its maximum distance from O_j, that is, $dist_{max}(O_j, p)$, shorter than its minimum distance from O_i, that is, $dist_{min}(O_i, p)$. On the other hand, if p is on the opposite side of $E_i(j)$, then $dist_{max}(O_j, p) \geq dist_{min}(O_i, p)$.

The UV-edge allows us to decide whether an object is an *answer object* (i.e., an object with nonzero qualification probabilities). In Fig. 3, q_0 is on the right of $E_i(j)$, which is also closer to O_j than O_i. Thus, $dist_{max}(O_j, q_0) <$

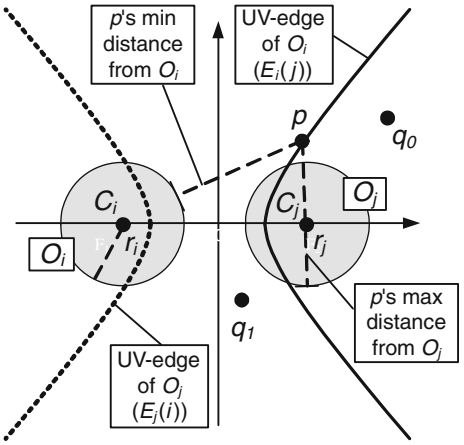


Fig. 3 The UV-edge

$dist_{min}(O_i, q_0)$. In other words, O_j is *always* closer to q_0 than O_i, and O_i has no chance to be the nearest neighbor of q_0. As another example, q_1 is on the left of $E_i(j)$. Since $dist_{min}(O_i, q_1) \leq dist_{max}(O_j, q_1)$, O_i has a nonzero qualification probability. Hence, given $E_i(j)$, if the query point is on the right of $E_i(j)$, O_i can be pruned.

We can now present a simple method for constructing a UV-cell. Let us define the *outside region*:

Definition 2 The outside region of UV-edge $E_i(j)$, denoted by $X_i(j)$, is the region on one side of $E_i(j)$ such that for any point $q \in X_i(j)$, O_j is always closer to q than O_i. We call the complement of $X_i(j)$ the inside region, denoted by $\overline{X_i(j)}$.

Given a set of n objects O, the UV-cell U_i of object O_i is essentially the intersection of all other $n-1$ inside regions:

$$U_i = \bigcap_{j=1...|O| \wedge j \neq i} \overline{X_i(j)} \tag{2}$$

Definition 3 A possible region of object O_i, denoted by P_i, is the intersection of a set of inside regions:

$$P_i = \bigcap_{j=1...|R| \wedge j \neq i \wedge R \subseteq O} \overline{X_i(j)} \tag{3}$$

According to the definition, the possible region should be an area that completely covers the UV-cell of O_i. An example of an object's possible region is the domain D, since D must cover any UV-cell. Here, R is the empty set. Notice that a UV-cell is also a possible region.

In Fig. 3, the outside region of the UV-edge $E_i(j)$ is the area on the right of the solid line. Notice that since q_0 is in the outside region of $E_i(j)$, O_j is closer to q_0 than O_i, and thus O_i cannot be q_0's nearest neighbor.

Given an object O_i, if we know all the outside regions $X_i(j)$ (where $j = 1, \ldots, n \wedge j \neq i$), then O_i's UV-cell can be constructed by excluding all these regions from D.

Algorithm 1 Generating a UV-cell

Input: Uncertain objects $O = \{O_1, O_2, \ldots, O_n\}$
Output: $U_1, U_2, \ldots, U_n$
1: **for each** $O_i \in O$ **do**
2: $P_i \leftarrow D$;
3: **for each** $O_j \in O \wedge j \neq i$ **do**
4: $E_i(j) \leftarrow$ UV-edge of O_i w.r.t. O_j;
5: $X_i(j) \leftarrow$ outside region of $E_i(j)$;
6: $P_i \leftarrow P_i - X_i(j)$;
7: **end for**
8: $U_i \leftarrow P_i$;
9: **end for**
10: **return** $U_1, U_2, \ldots, U_n$

Algorithm 1 illustrates the basic method for constructing UV-cell for n objects. The possible region of each object O_i is first initialized as the whole space (Step 2). Then, for each O_j, we compute the UV-edge of O_i and its corresponding outside region (Steps 4 and 5). The possible region, which contains all the points that may have O_i as one of their nearest neighbors, is then "reduced" by the outside region that overlaps with it (Step 6). The UV-cell of O_i is then assigned to be the final possible region (Step 8).

We now discuss Step 6 in more detail. This step uses $E_i(j)$ to "refine" the edges of P_i, that is, find the intersections of $E_i(j)$ with P_i. Specifically, for each UV-edge e of P_i, we compute the intersection of $E_i(j)$ and e. Since $E_i(j)$ is a hyperbola, we can use the techniques in [3] to do this. The resulting intersections partition $E_i(j)$ into some segments. For each segment s, there are two scenarios:

- Case 1: s is inside P_i: We refine P_i by using s as one of the new edges of P_i. Some existing edges of P_i are removed if necessary.
- Case 2: s is outside P_i (except the end points of s): P_i cannot be changed by s, and we do not have to do anything to handle this case.

After we have visited all the segments of $E_i(j)$, we have found all the intersections of $E_i(j)$ and P_i. Moreover, P_i is refined, and Step 6 is completed.

As an example of Step 6, consider Fig. 4a, which illustrates P_i, and Fig. 4b, which shows the result of intersecting $E_i(j)$ and P_i. The segment between v_4 and v_5 is inside P_i. Since *Case 1* is satisfied, the existing edges between v_4 and v_5 (i.e., (v_4, v_1') and (v_1', v_5)) are replaced by segment (v_4, v_5). On the other hand, (v_5, v_6) is outside P_i, and so Case 2 is satisfied. There is no need to change any edges of P_i between v_5 and v_6. The process is repeated until all the segments are visited. As shown in Fig. 4b, vertices v_1', v_2', and v_3' are removed from P_i, while $v_4, \ldots, v_9$ are added to it.

Note that the order of selecting the object for refining O_i's possible region (Steps 4–6) does not affect the correctness of the algorithm; the UV-cell is produced by "shrinking" the possible regions by using the outside regions of other objects.

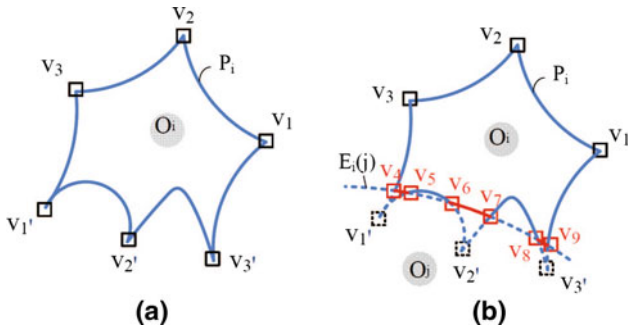


Fig. 4 **a** Before checking $E_i(j)$. **b** After checking $E_i(j)$

Also, not all objects are useful in shaping a UV-cell. In Sect. 5, we will explain how to prune away these objects.

4.2 The shape of a UV-cell

We now study a mathematical representation of the UV-cell. We also derive the number of UV-edges of a given UV-cell. Here, we assume that the uncertainty region of O_i is a circle, with center c_i and radius r_i, with $r_i > 0$. Later, we discuss the UV-cell of a "point uncertainty" (i.e., $r_i = 0$), and also uncertainty regions that are not circle in shape.

For any point $q \in D$, we can observe from Fig. 3 that:

$$dist_{min}(O_i, q) = \begin{cases} dist(q, c_i) - r_i & \text{if } q \notin \odot(c_i, r_i) \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$$dist_{max}(O_j, q) = dist(q, c_j) + r_j \quad (5)$$

where $\odot(c_i, r_i)$ denotes a circle with center c_i with radius r_i. Since $r_j > 0$, $dist_{max}(O_j, q)$ must also be positive. By substituting Eqs. 4 and 5 into Eq. 1, we have:

$$dist(q, c_i) - dist(q, c_j) = r_i + r_j \quad (6)$$

Let the coordinates of c_i and c_j be (x_i, y_i) and (x_j, y_j). Let $f_x = \frac{1}{2}(x_i + x_j)$ and $f_y = \frac{1}{2}(y_i + y_j)$. Let $\cos\theta = \frac{(x_j - x_i)}{dist(c_i, c_j)}$ and $\sin\theta = \frac{(y_j - y_i)}{dist(c_i, c_j)}$. Then, Eq. 6 becomes:

$$\frac{x_\theta^2}{a^2} - \frac{y_\theta^2}{b^2} = 1 \quad (7)$$

where

- $a = \frac{r_i + r_j}{2}$, $c = \frac{dist(c_i, c_j)}{2}$, and $b = \sqrt{c^2 - a^2}$;
- $x_\theta = (x - f_x)\cos\theta + (y - f_y)\sin\theta$;
- $y_\theta = (f_x - x)\sin\theta + (y - f_y)\cos\theta$.

Essentially, Eq. 7 is a hyperbolic equation, with c_i and c_j as the foci, rotated by θ in an anti-clockwise sense [3]. Figure 3 illustrates that the UV-edge of O_i w.r.t. O_j (the solid line) is a hyperbola.

Equation 7 shows that a UV-cell is composed of the intersections of one or more UV-edges, which are hyperbolas. Since a hyperbola is a conic curve, an UV-edge must be *concave* in shape. In Fig. 2, apart from the edges of the domain space, the UV-cells of the three objects have concave edges. Note that Eq. 7 has two curves, which represent the UV-edges for each pair of objects involved. For example, in Fig. 3, the solid line is the UV-edge of O_i w.r.t. O_j, and the dotted line is the UV-edge of O_j w.r.t. O_i.

If two objects overlap, then $dist(c_i, c_j) < r_i + r_j$, and in Eq. 7, b is not a real number. Physically, this means $E_i(j)$ cannot be found, and we can treat $X_i(j)$ as an empty region.

We now revisit Algorithm 1. Step 4 is done using Eq. 7. Step 5 is performed by observing that the outside region of a UV-edge must be convex in shape. To perform Step 6 (i.e., cutting the possible region by an outside region), we compute the intersections of hyperbola equations using linear algebra techniques [3], which are detailed in Appendix 9.

Let us state an interesting observation about a possible region, which we will use later.

Lemma 1 *The possible region of an uncertain object is a connected region without any hole inside it.*

The proof of this lemma, detailed in Appendix 10, shows that a contradiction will result if a possible region contains a hole. We next discuss the shape of the UV-cell for other kinds of uncertainty regions.

(1) Point uncertainty. Given two objects O_i and O_j, suppose that at least one of them has no uncertainty, i.e., r_i or r_j is equal to zero. There are two scenarios:

- If $c_i \neq c_j$, without loss of generality, assume that $r_i = 0$. Then, $E_i(j)$ can be obtained by Eq. 7, because all variables used in that equation are real numbers, and a, b are nonzero values. Notice that $E_i(j)$ becomes a perpendicular line segment when $r_i = r_j = 0$.
- If $c_i = c_j$, then $E_i(j)$ does not exist. If $r_i \neq r_j$, Eq. 1 does not hold, and the UV-cell of O_i, or U_i, does not exist. If $r_i = r_j$, Eq. 1 always holds, and $U_i = D$.

(2) Non-circular uncertainty regions. To find the UV-cells for non-circular uncertainty regions, our first attempt is to derive the UV-edges for objects with rectangular uncertainty regions. As shown in Fig. 5, the UV-edge between objects O_1 and O_2 is a piecewise-quadratic line segments. This is too expensive to compute and store. Instead, for each object O_i, we convert its non-circular uncertainty region to a circle, $MBC(O_i)$, which minimally contains it. Then, we use Algorithm 1 to construct the UV-cells for these circles. We claim that $MBC(O_i)$'s UV-cell always covers that of O_i.

To understand why, notice that if some object O_1 may be q's nearest neighbor, then $MBC(O_1)$ can also be q'snearest

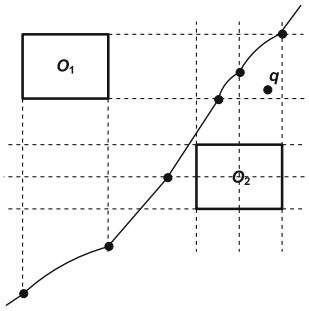


Fig. 5 A UV-Edge for rectangular regions

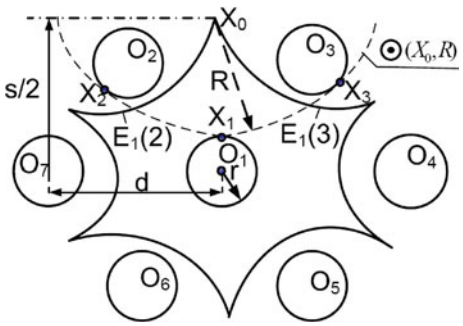


Fig. 6 Estimating the size of a UV-cell

neighbor. First, for all $i = 1, \ldots, n, dist_{min}(q, O_1) < dist_{max}(q, O_i)$. Also,

$$\begin{cases} dist_{min}(q, MBC(O_i)) < dist_{min}(q, O_i), \\ dist_{max}(q, MBC(O_i)) > dist_{max}(q, O_i) \end{cases}$$

Hence, $dist_{min}(q, MBC(O_1)) < dist_{min}(q, O_1)$, which is less than $dist_{max}(q, O_i)$, and is less than $dist_{max}(q, MBC(O_i))$. Therefore, $MBC(O_1)$ is q's possible nearest neighbor, among $\{MBC(O_i)\}_{i=1}^{n}$. If q is situated in the UV-cell of O_1, it must also be located in the UV-cell of $MBC(O_1)$. In other words, $MBC(O_i)$'s UV-cell always cover that of O_i. Therefore, in answering a PNN, if we found that $MBC(O_i)$ contains q, we have to verify whether O_i can be the nearest neighbor of q. In the sequel, we assume that all uncertainty regions are circular.

4.3 The number of UV-edges of a UV-cell

Let us now examine the number of UV-edges of a UV-cell. As Algorithm 1 shows, for every object O_i, its UV-edge with respect to other objects is used to refine its possible region P_i (Step 6). This requires computing the intersections of all edges of P_i with a new UV-edge $E_i(j)$, for some object O_j.

As shown in Fig. 4b, $E_i(j)$ intersects with U_i's UV-edge (v'_1, v'_2) at v_5 and v_6. Thus, (v'_1, v'_2) is replaced by three edges: (v_4, v_5), (v_5, v_6), and (v_6, v_7).

From this example, we can see that $E_i(j)$, a hyperbolic curve, can have at most 2 intersections with a UV-edge of P_i; and 3 new edges can be created for P_i as a result. In the worst case, the number of edges of P_i increases by 3 times whenever a new edge is considered.

In general, to obtain U_i, we have to take into account $n-1$ objects. Hence, the number of edges of the UV-cell has an (exponential) upper bound of $O(3^n)$.

Moreover, computing intersections between hyperbolas is complex. In our implementation, 60 h are needed to create a UV-diagram of 40,000 objects by using Algorithm 1 . We will explain how to find an efficient representation of the UV-cell, in Sect. 5.

4.4 The size of a UV-cell

We now estimate the size of a UV-cell, under the assumption that all objects are evenly placed. We consider the *hexagonal lattice model*, where each object has six neighbors whose centers are equidistant from each other, with distance d_0.[1] We assume that the uncertainty region sizes of all objects are the same, with a radius of r. Figure 6 illustrates seven objects configured in this manner. Given an object O_1, we assume that the UV-cell U_1 of O_1 is not trimmed by the boundary of the domain space. That is, its UV-cell is solely determined by the uncertainty regions of other objects. Our goal is to find the dimension s of a square that contains U_1. This square should be a good approximation of U_1.[2]

Let $H(d)$ be a set of six objects whose uncertainty region's centers have the same distance d from that of O_1. For example, Fig. 6, the centers of the uncertainty regions of $H(d) = \{O_2, \ldots, O_7\}$ are d units away from that of O_1. We claim the following:

Lemma 2 *Let $P_{1,d}$ be the possible region of O_1 generated by the objects in $H(d)$. The length of the square, which is centered at c_1 and minimally covers $P_{1,d}$, is as follows:*

$$s(d) = \frac{2d^2 - 8r^2}{\sqrt{3}d - 4r} \quad if d > \frac{4r}{\sqrt{3}} \tag{8}$$

In the sequel, we use $s(d)$ to denote the size of $P_{1,d}$. The proof of Lemma 2 can be found in Appendix 11. Notice that $P_{1,d}$ contains U_1.

Now, observe that the centers of the six objects in $H(d_0)$ form the vertices of a hexagon called HEX_1. This hexagon is illustrated in Fig. 7. As shown in [35], a larger hexagon

[1] The centers of uncertainty regions form the vertices of n hexagons, each of which has an area of $\frac{\sqrt{3}d_0^2}{2}$. Since $|D| = n \times \frac{\sqrt{3}d_0^2}{2}$, $d_0 = \sqrt{\frac{2|D|}{\sqrt{3}n}}$.

[2] As shown in Fig. 2, a UV-cell can be irregular in shape, and so estimating its size is not easy. Thus, we use a simple data model here. We will also explain how these results can be applied to uniformly distributed data, in Sect. 5.2.

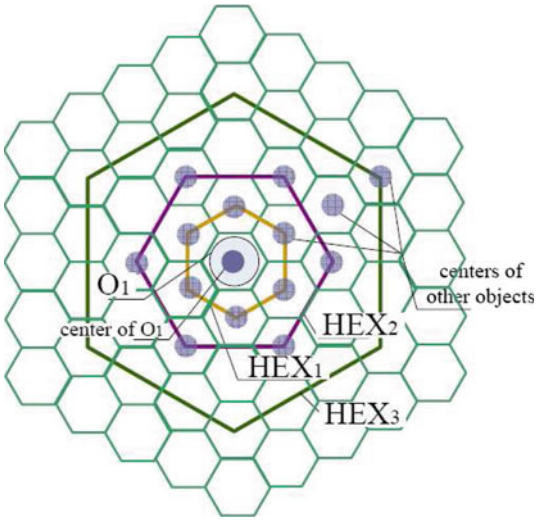


Fig. 7 Illustrating O_1 and its neighbors

HEX_{i+1}, formed by the centers of six other objects, can be obtained by rotating HEX_i by $\frac{\pi}{6}$ radians, and scaling it by a factor $\sqrt{3}$. Figure 7 shows how HEX_2 and HEX_3 are generated in this way. We then obtain the following result.

Theorem 1 *If $d_0 > 2\sqrt{3}r$, then the square that minimally contains U_1 has a size of $s(d_0)$ obtained from Eq. 8.*

The main idea of the proof is that when $d_0 > 2\sqrt{3}r$, the six objects that form HEX_1 alone contribute to the edges of O_1's UV-cell. Its details can be found in Appendix 12.

An iterative approach of finding d^*. We now explain how to derive the size of a square that contains U_1, for any value of d_0. Our goal is to find d^* among different values of d, such that the square covering P_{1,d^*} is the smallest. We observe from the first-order derivative of s from Eq. 8 that $d^+ = 2\sqrt{3}r$ is the only inflexion point, such that s monotonously decreases when $\frac{4r}{\sqrt{3}} < d < d^+$, and monotonously increases when $d \geq d^+$. However, this result cannot be readily used, since we may not be able to find six neighbors of O_1 that are exactly d^+ units apart from each other. We thus estimate d^* as follows. We first consider the objects on HEX_1, and compute $s(d_0)$. We then consider HEX_2, where each vertex is $\sqrt{3}d_0$ from c_1, and evaluate $s(\sqrt{3}d_0)$. We repeat this process, until the six objects found are d_x units apart from each other, where (1) $d_x > \frac{4r}{\sqrt{3}}$ and (2) $s(d_x) < s(\sqrt{3}d_x)$. Then, we set $d^* = d_x$, and use Lemma 2 to find s.

The above process only examines the values of d at $d_0, \sqrt{3}d_0, (\sqrt{3})^2 d_0, \ldots, \sqrt{|D|}$. Hence, at most $\left\lceil log_{\sqrt{3}}(\sqrt{|D|}/d_0) \right\rceil$ trials are needed to find d^*. Although this procedure does not find the square that tightly contains a UV-cell, our experiments show that the approximation is highly accurate.

5 Efficient UV-cell generation

Since generating a UV-cell is inefficient, our strategy is to avoid computing it directly. Instead, we represent a UV-cell as a set of *candidate reference objects* (cr-objects), which can be efficiently derived. As will be discussed in Sect. 6, cr-objects can be used to develop an approximate representation of the UV-diagram. Section 5.1 outlines the algorithm of yielding cr-objects. We explain the preparation phase of this algorithm as well as two techniques for finding these objects quickly, in Sects. 5.2 and 5.3, respectively. Section 5.4 discusses how to derive cr-objects efficiently for a group of nearby objects.

5.1 Reference objects and candidate reference objects

Recall from Algorithm 1 that the UV-cell of an object O_i, that is, U_i, is the result of repeatedly subtracting the outside region of other objects (i.e., $X_i(j)$) from its possible region, P_i. In fact, not all outside regions are useful for refining P_i. In particular, if the UV-edge of O_i corresponding to O_j, that is, $E_i(j)$, does not intersect with P_i, then P_i cannot be shrinked by X_i(j). We call an object O_j a *reference object* (or *r-object*) of O_i, if O_j defines an edge of O_i's UV-cell. We also denote $F_i \subseteq O$ to be the set of r-objects of O_i. The set F_i contains objects whose outside regions are responsible for defining the UV-cell of O_i. In Fig. 2, for example, the set of r-objects of O_3, that is, F_3, is $\{O_1, O_2\}$.

Given that the r-objects for each object are known, our solution (to be shown in Sect. 6) can use r-objects to develop an alternative representation of the UV-diagram. This solution is much cheaper than Algorithm 1 , which requires exact UV-cells to be computed. However, finding F_i itself is difficult because we do not know the UV-cell of O_i. Our strategy is to find a small set C_i of objects, where $F_i \subseteq C_i$. We call C_i the *candidate reference objects* (or *cr-objects* in short). We next show how C_i can be derived without acquiring the exact UV-cell of O_i. In Sect. 6, we study an indexing solution based on cr-objects.

Algorithm 2 (`getcrObject`(O_i, S)) presents a procedure that derives the cr-object set C_i for object O_i, based on a set S of objects. To retrieve C_i, we can simply invoke `getcrObject`(O_i, O). In this algorithm, Step 1 (`initPossibleRegion`) creates a possible region P_i based on a small number of objects retrieved from S. In Step 2, the "index level" pruning (or `iPrune`) yields a set I of objects that may contribute edges to the UV-cell. Step 3 applies "computational level" pruning (or `cPrune`) on I, and produces C_i. Here, we assume that an R-tree index has been built on the uncertainty regions of the objects in O. Each object's information (e.g., uncertainty region and pdf) is stored in the disk. Next, we explain how to generate an initial possible region (Sect. 5.2), based on which two techniques for pruning non-cr-objects are developed (Sect. 5.3).

Algorithm 2 `getcrObject`(O_i, S)

Input: Uncertain object O_i
Output: cr-object C_i
1: $P_i \leftarrow$ `initPossibleRegion`(O_i, S)
2: $(P_i, I) \leftarrow$ `iPrune`(P_i)
3: $C_i \leftarrow$ `cPrune`(P_i, I)

5.2 Generating a possible region

In Step 1 of Algorithm 2 , we retrieve a small number of objects, called *seeds*, from a set of objects S. These seeds are used to generate an "initial" possible region, using a routine similar to Steps 3–7 of Algorithm 1. This region is used by other pruning methods to produce cr-objects.

Seeds have to be selected with care. If seeds are randomly selected, a big initial region can be produced. This region may be intersected by many outside regions, resulting in a poor pruning efficiency. Ideally, we would like the initial possible region generated by these seeds to closely resemble the UV-cell. We would also prefer the number of these seeds to be small, so that the possible region can be constructed efficiently. We next present two simple steps to find "good" seeds.

Step (i). We issue a k-Nearest-Neighbor Query (k-NN) on S, by using the center c_i of O_i's uncertainty region as the query point. The k objects, which are not O_i and whose regions' minimum distances from c_i are the shortest, are obtained. Since these objects are close to O_i, we consider them to have a good chance for defining the UV-edges of U_i. They are thus good candidates for being seeds. Note that if $S = O$, then the R-tree on O can be used to support the k-NN search.
Step (ii). Out of the k objects obtained from Step (i), we select k_s seeds. These objects are chosen in way such that they are evenly spread, in order to generate a "good" possible region. In particular, we divide the domain D into k_s equally sized sectors, centered at c_i. For each sector, the object closest to c_i is a seed.

The above method does not guarantee that all k_s seeds can be found (e.g., no seeds can be found if a sector is empty). Even if this happens, however, we can still obtain an initial possible region without affecting the latter steps. This region may be larger, though. In our experiments, $k_s = 30$, and in most cases, all seeds can be found. For each object, evaluating a k-NN query requires $O(|S|)$ time, selecting seeds costs $O(k)$ time, and constructing an initial region needs $O(k_s)$ time. Hence, the cost of this step is $O(|S| + k + k_s)$.

Model-based seed selection. We can use the results in Sect. 4.4 to estimate the value of k derived from *Step (i)*. We assume that all the objects in domain D follow the hexagonal lattice model. First, we find the size s of the square that bounds the UV-cell of O_i. Particularly, we check whether the condition for Theorem 1 is satisfied. If this is true, we let $d_{min} = d_0$. Otherwise, we use the iterative approach, described in Sect. 4.4, to find d^*, and let $d_{min} = d^*$. Then, we find $s(d_{min})$ by using Lemma 2. Figure 6 shows that the maximum distance of any point on the possible region P_i from the center c_i of O_i's uncertainty region is $\frac{s}{2}$. If we draw a circle of radius $(s - r)$, centered at c_i, then Theorem 3 (to be discussed in Sect. 5.3) tells us that only objects located in this circle can be the reference objects F_i. We can then estimate k as the expected number of objects in $\odot(c_i, s - r)$:

$$k = \lceil \pi (s - r)^2 \rho \rceil \tag{9}$$

We can also use the above approach in a database whose locations are uniformly distributed in D. We first compute the average uncertainty radius r_a of these objects. We then suppose that all these objects have the same radius r_a. We also evaluate the distance of each object from its nearest neighbor, and find the average d_a of these distances. The values of the radius r and the distance d_0 of the hexagonal model are set to be r_a and d_a, respectively. We also compute the density ρ, which is equal to $\frac{\text{No. of objects in} D}{\text{Area of } D}$. Our experiments show that this model can enhance the seed selection process for uniformly distributed data.

5.3 I-Pruning and C-Pruning

Once the possible region has been initialized, we perform *I-pruning* and *C-pruning* (Steps 2 and 3 of Algorithm 2), in order to remove objects that cannot constitute a UV-edge to the UV-cell. Let us now examine these two steps in more details.

Step 2: Index Level Pruning. To understand this step, let us consider an object O_i, its possible region P_i, and another object O_j, which has not yet been considered for refining P_i. Our goal is to establish the necessary and sufficient condition(s) for O_j to have effect on the shape of P_i. We first claim the following.

Lemma 3 *$P_i = P_i - X_i(j)$, if and only if for every point p inside P_i, $dist_{max}(p, O_j) > dist_{min}(p, O_i)$.*

Proof **(If)** For every $p \in P_i$, p cannot be on $X_i(j)$. If this is false, then O_j is always closer to p than O_i, i.e., $dist_{max}(p, O_j) \leq dist_{min}(p, O_i)$ (Definition 2). This violates the condition that $dist_{max}(p, O_j) > dist_{min}(p, O_i)$. Hence, $p \notin X_i(j)$, and $P_i = P_i - X_i(j)$.

(Only if) Suppose there exists a point p' inside P_i, such that $dist_{max}(p', O_j) \leq dist_{min}(p', O_i)$. Then O_j is always closer to p' than O_i, and O_i cannot be the nearest neighbor of p'. This implies that p' must be excluded from P_i after O_j is considered. Hence, P_i cannot be equal to $P_i - X_i(j)$, resulting in a contradiction.

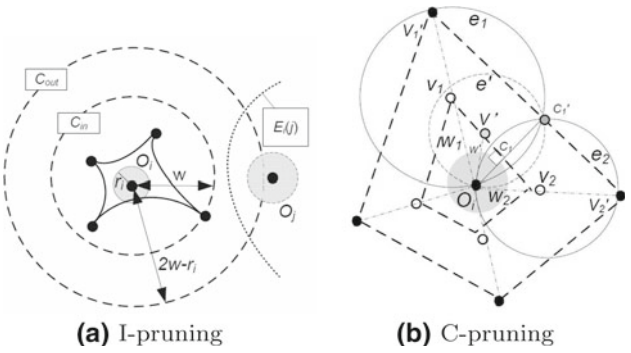


Fig. 8 Our pruning methods

Let $b(P_i)$ be the boundary of P_i. We have:

Theorem 2 *$P_i = P_i - X_i(j)$ if and only if for every point $p \in b(P_i)$, $dist_{max}(p, O_j) > dist_{min}(p, O_i)$.*

Proof **(If)** Let us first show that:

$$\forall p'' \in P_i, dist_{max}(p'', O_j) > dist_{min}(p'', O_i) \quad (10)$$

Suppose by contrary that the above is not correct. That is, $\exists p' \in P_i - b(P_i)$, such that $dist_{max}(p', O_j) \leq dist_{min}(p', O_i)$. If we let P_i' be $P_i - X_i(j)$, then $p' \in X_i(j)$ and $p' \notin P_i'$. From the given condition, we can see that for every $p \in b(P_i)$, $p \notin X_i(j)$, and $p \in P_i'$. Thus, P_i' must have a hole (p') inside it. However, this must not be true, according to Lemma 1.

Hence, Eq. 10 is true. Using Lemma 3, we see that $P_i = P_i - X_i(j)$, and the so the "if" part is correct.

(Only if) From Lemma 3, we know that for every point $p \in P_i, dist_{max}(p, O_j) > dist_{min}(p, O_i)$. Since $b(P_i) \subseteq P_i$, the "only if" part is correct.

Essentially, if we want to examine whether O_j has any effect on P_i, it suffices to consider the points on P_i's boundary, instead of all points in P_i. We next present the following theorem, which forms the basis of I-pruning.

Theorem 3 *Given an object O_i with center c_i and radius r_i, let w be the maximum distance of P_i from c_i. Let C_{out} be a circle, with center c_i and radius $2w - r_i$. For another object O_j, if $c_j \notin C_{\text{out}}$, then $P_i = P_i - X_i(j)$.*

Proof Denote C_{in} by a circle with center c_i and radius w. Figure 8(a) illustrates O_i, its possible region P_i (in solid lines), C_{in} and C_{out}. Let us suppose on the contrary that P_i is not equal to $P_i - X_i(j)$, that is, P_i can be reshaped by the UV-edge of O_j. Then, using Theorem 2, there must exist a point p on the boundary of P_i such that:

$$dist_{max}(p, O_j) \leq dist_{min}(p, O_i) \quad (11)$$

Using Eqs. 4 and 5, we have:

$$dist(p, c_j) + r_j \leq dist(p, c_i) - r_i$$
$$\Rightarrow dist(p, c_j) + dist(p, c_i) + r_j \leq 2 \cdot dist(p, c_i) - r_i$$
$$\Rightarrow dist(p, c_j) + dist(p, c_i) \leq 2 \cdot dist(p, c_i) - r_i$$
$$\Rightarrow dist(c_i, c_j) \leq 2 \cdot dist(p, c_i) - r_i \quad (12)$$

since $dist(c_i, c_j) \leq dist(p, c_j) + dist(p, c_i)$ due to the triangular inequality. Now, $dist(p, c_i) \leq w$, so Eq. 12 becomes:

$$dist(c_i, c_j) \leq 2w - r_i \quad (13)$$

This implies that c_j is in the circle C_{out}, contradicting the assumption of Theorem 3. Hence, this lemma is correct.

The I-pruning method uses Theorem 3 by issuing a circular range query, centered at c_i with radius $2w - r_i$, on the dataset O. This operation can be easily implemented by using the R-tree created for O. The range query first uses the R-tree to filter all objects that do not overlap with the range. For the remaining objects, they are removed if their centers are beyond the circular range. Hence, in this phase (Step 2 of Algorithm 2), a cost of $O(n)$ is needed.

Step 3: Computational Level Pruning.
We next discuss a method, based on distance comparison, to check whether object O_j can affect the possible region of object O_i. We call this method *C-pruning* (Step 3 of Algorithm 2). Theorem 4, discussed below, serves as the foundation of C-pruning.

Theorem 4 *Given an uncertain object $O_i(c_i, r_i)$ and P_i's convex hull $CH(P_i)$, let $v_1, v_2, \ldots, v_n$ be $CH(P_i)$'s vertex. If another object O_j's center c_j is not in any of $\{\odot(v_m, dist(v_m, c_i))\}_{m=1}^{n}$, then $P_i = P_i - X_i(j)$.*

Proof First, the convex hull $CH(P_i)$, which completely contains P_i, must also be O_i's possible region. For every point p on $CH(P_i)$'s boundary, suppose c_j is located outside the circle $\odot(p, dist(p, c_i))$. Then, we have:

$$dist(p, c_j) > dist(p, c_i)$$
$$\Rightarrow dist(p, c_j) + r_j > dist(p, c_i) - r_i$$
$$\Rightarrow dist_{max}(p, O_j) > dist_{min}(p, O_i) \quad (14)$$

Second, Theorem 2 states that if $dist_{max}(p, O_j) > dist_{min}(p, O_i)$, then $CH(P_i) = CH(P_i) - X_i(j)$. Therefore, if c_j is outside $\odot(p, dist(p, c_i))$ for every p on $CH(P_i)$'s boundary, O_j can be safely pruned.

For convenience, let $\odot(p, dist(p, c_i))$ be a w-bound (where $w = dist(p, c_i)$). We also define a set S of w-bounds for every point p in U_i. We now show that instead of checking all the w-bounds in S, it is only necessary to check those w-bounds constructed for the vertices of $CH(P_i)$. Specifically, the w-bounds of the vertices must contain all other w-bounds of all points on the boundary of $CH(P_i)$. To see this, let w_k be the distance of vertex v_k from O_i's center. We extend each vertex v_k by the distance w_k to obtain a new

vertex v'_j (black dot in Fig. 8b). These new vertices are connected to form a polygon. We use e_1 and e_2 to represent the w-bounds $\odot(v_1, w_1)$ and $\odot(v_2, w_2)$, respectively.

We next show that, for any point v' on $CH(P_i)$'s edge v_1v_2, $\odot(v', dist(v', c_i)) \subseteq e_1 \cup e_2$, where we let $e' = \odot(v', dist(v', c_i))$. We draw a line $c_1c'_1$, which is perpendicular with v_1v_2 and $v'_1v'_2$, and intersects them at points c_1 and c'_1, respectively. As v_1v_2 is the perpendicular bisector of $c_ic'_1$, we see that $c_ic'_1$ is the common chord of e_1, e_2 and e'. Since e_1 or e_2 is bigger than e', e' is contained by e_1 or e_2.

Hence, to check whether O_j can refine P_i, we just need to check the set of w-bounds $S' = \{\odot(v_m, dist(v_m, c_i))\}$ (where $S' \subseteq S$). If c_j is located outside all w-bounds in S', then $CH(P_i) = CH(P_i) - X_i(j)$. Finally, since P_i is completely covered by $CH(P_i)$, $P_i = P_i - X_i(j)$ must also be true. This completes the proof.

Step 3 of Algorithm 2 uses Theorem 4 to prune unqualified objects returned by I-pruning. This can be done efficiently, because only the vertices of $CH(P_i)$ are used. Moreover, $|CH(P_i)|$ is small, since the possible region is only derived by a small number k_s of seeds. The complexity of this phase is $O(n)$.

We consider the objects that remain after this step as cr-objects (i.e., C_i). The complexity of Algorithm 2, for generating C_i's of n objects, is $O(n(n + k))$.

5.4 Batch processing of cr-objects

To create the UV-index, we first find out the cr-objects for each of the n database objects. A simple way to do this is to run Algorithm 2 (i.e., `getcrObject`(O_i, O)) for all objects $O_i \in O$, as proposed in [17]. However, this involves running `getcrObject` for n times and can be quite costly. We now present a *Batch Processing* algorithm (or `BP` in short), where the cr-objects of a group of objects are considered together. As we will show, this new algorithm allows the effort of devising an object's cr-objects to be shared by others, and consequently reduces a lot of cr-object generation overhead.

Observe that if an object O_i is near to O_j, then their UV-cells should be similar. The cr-object set of O_i, that is, C_i, can then be similar to C_j. The `BP` makes use of this principle; it employs C_i to derive C_j, instead of generating C_i and C_j independently. Let G be a set of objects that are physically close to each other. The `BP` first computes a set of objects C_G, a superset of C_i, for every $O_i \in G$. The cr-objects of objects in G are then extracted from C_G. Usually, C_G is smaller than the database size $|O|$, and thus retrieving cr-objects from C_G is faster than from O.

Algorithm 3 presents the `BP`. Given $G \subseteq O$, Step 1 creates a new object O_G. The uncertainty region of O_G is the minimum bounding circle (MBC) of the uncertainty regions

Algorithm 3 `BP`

Input: A set G of objects in O
Output: cr-object set C_i for each $O_i \in G$
1: $O_G \leftarrow (MBC(G), \textit{uniform pdf})$
2: $C_G \leftarrow$ `getcrObject`(O_G, O)
3: **for each** object $O_i \in G$ **do**
4: $P_i \leftarrow$ `initPossibleRegion`(O_i, C_G)
5: $C_i \leftarrow$ `cPrune`(P_i, C_G)
6: **end for**

of all objects in G. Its uncertainty pdf is not important here, and we assume it to be uniform. Notice that O_G is only used by the `BP`; it will be deleted after the algorithm halts.

Step 2 invokes a slightly modified version of `getcrObject` to obtain a cr-object set C_G of O_G. Particularly, in Step (i) of `initPossibleRegion`, the k-NN search skips all objects in G. Notice that `initPossibleRegion` computes the possible region of an object. In Step (i) of that procedure, we obtain the seeds – objects that are useful for generating a UV-cell. In Algorithm 3, the input of `getcrObject` is O_G, whose uncertainty region includes the uncertainty regions of all objects in G. Therefore, the uncertainty region of any object $O_i \in G$ overlaps with that of O_G. More importantly, $O_i \in G$ is not useful for finding possible regions of O_G, because O_i does not create any UV-edge for O_G's UV-cell. We next claim the following:

Theorem 5 *Given an object O_j, if $O_j \notin C_G$ after Step 2 of Algorithm 3, then $O_j \notin F_i$, where $O_i \in G$.*

That is to say, any object not contained in C_G cannot be an r-object of $O_i \in G$. In other words, C_G is a superset of r-objects for all the objects in G. The proof of this theorem, which is quite complex, is detailed in Sect. 5.5. Notice that all objects in G are included in C_G after the execution of Step 2. This is because in the last step of `getcrObject` (Algorithm 2), objects whose centers are located in the c-pruning bound of O_G are treated as cr-objects. Since the center of an object in G is inside O_G's c-pruning bound, it must also be a cr-object of O_G. Thus, $G \subseteq C_G$.

Steps 3–6 use C_G to generate cr-objects for each object $O_i \in G$. From Theorem 5, we know that an object in C_G may be an r-object of O_i. Thus, objects in C_G can be considered as good candidates for generating an initial possible region, P_i for O_i. We thus pass C_G to `initPossibleRegion` and get P_i (Step 4). We then execute `cPrune` on C_G to retrieve C_i (Step 5). These two steps are repeated for all objects in G, until we obtain their cr-objects.[3]

The `LP` algorithm. We now discuss a way to use Algorithm 3 to construct cr-objects for O. The *Leaf-Node*

[3] We do not execute `iPrune`(P_i, O) after Step 4 because the set of objects returned by `iPrune` is often the superset of C_G in our experiments. Thus, `iPrune` is not very effective here.

Processing, or LP, performs a preorder traversal of the R-tree that indexes O. When a leaf node, say N, is reached, BP is invoked on all objects stored in N, in order to compute their cr-objects. The algorithm terminates when all leaf nodes have been exhausted.

The LP can generate cr-objects for O quickly. This is because when the BP is called, it always uses the objects located in a leaf node. In an R-tree, the leaf node consists of a set G of objects, which are physically close to each other. Recall that the object created in Step 1 of BP (i.e., O_G) is the MBC of the uncertainty regions of objects in G. Thus, the size of O_G would not be very different from those of the objects in G. Consequently, the set C_G derived from Step 2 (`getcrObject`) should also be similar to the r-objects of G's objects. In our experiments, $|C_G|$ is much smaller than $|O|$. Hence, Step 4 can be carried out more efficiently than if `initPossibleRegion` is carried out on O for every object.

We have introduced an efficient construction method to derive the cr-object set C_i for O_i. We have also explained how to obtain cr-objects for O quickly. One may consider to use C_i to generate the exact UV-cell of O_i. However, our experiments show that $|C_i|$ may be large, and so generating the UV-cell of O_i can still be costly. In Sect. 6, we show how to use C_i directly to construct an index for the UV-diagram. In the rest of this section, we present the proof of Theorem 5.

5.5 Proof of Theorem 5

Recall that O_G is formed by a set G of objects, using Step 1 of Algorithm 3. Let $P_i(S)$ be a possible region of an object O_i, constructed by using a set $S \in O$ of objects. Essentially, $P_i(S)$ is the intersection of the inside regions $\overline{X_i(k)}$, where $O_k \in S$. Let u_i be the uncertainty region of O_i. We first claim the following.

Lemma 4 *Given a set S of objects, where $S \subseteq O$, for any objects O_i and O_k, if $u_i \subseteq u_k$, then $P_i(S) \subseteq P_k(S)$.*

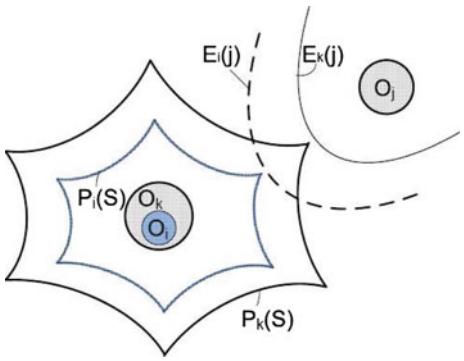


Fig. 9 Illustrating Lemmas 4 and 6

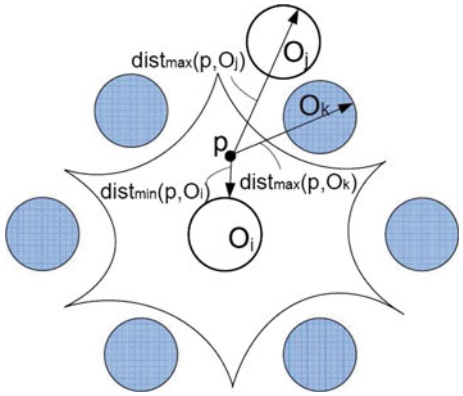


Fig. 10 Illustrating Lemma 5

Figure 9 illustrates Lemma 4, which shows that $P_i(S)$ is inside $P_k(S)$. Its proof can be found in Appendix 13.1.

Lemma 5 *Given objects O_i and O_j, if $c_j \notin P_i$, then $\forall p \in P_i(S)$:*

$$dist_{max}(p, O_j) > dist_{min}(p, O_i) \tag{15}$$

if and only if $\exists O_k \in S$, *where*

$$dist_{max}(p, O_j) > dist_{max}(p, O_k) \tag{16}$$

In Fig. 10, the objects in S are shaded. The center of O_j, that is, c_j, is outside $P_i(S)$. Given a point $p \in P_i$, Lemma 5 states that if there is an object $O_k \in S$ such that $dist_{max}(p, O_j) > dist_{max}(p, O_k)$, then $dist_{max}(p, O_j) > dist_{min}(p, O_i)$, or vice versa. Its proof can be found in Appendix 13.2. These results are used by the next lemma.

Lemma 6 *Given two objects O_i and O_k, where $u_i \subseteq u_k$, and an object O_j where $c_j \notin P_k(S)$, if $P_k(S) = P_k(S) - X_k(j)$, then $P_i(S) = P_i(S) - X_i(j)$.*

As shown in Fig. 9, Lemma 6 claims that given an object O_j whose center is outside $P_k(S)$, if the edge $E_k(j)$ does not affect the possible region $P_k(S)$, then $E_i(j)$ cannot contribute to $P_i(S)$.

Proof Since $P_k(S) = P_k(S) - X_k(j)$, by using Lemma 3, we have:

$$\forall p \in P_k(S), dist_{max}(p, O_j) > dist_{min}(p, O_k) \tag{17}$$

Using the "only-if" part of Lemma 5, we have:

$$\forall p \in P_k(S), \exists O_t \in S, dist_{max}(p, O_j) > dist_{max}(p, O_t) \tag{18}$$

Since $u_i \in u_k$, using Lemma 4, we have $P_i(S) \in P_k(S)$. Thus, Eq. 18 becomes:

$$\forall p \in P_i(S), \exists O_t \in S, dist_{max}(p, O_j) > dist_{max}(p, O_t) \tag{19}$$

Using the "if" part of Lemma 5, Eq. 19 becomes:

$$\forall p \in P_i(S), dist_{max}(p, O_j) > dist_{min}(p, O_i) \quad (20)$$

Using Lemma 3, Eq. 20 means that $P_i = P_i - X_i(j)$. Hence, the lemma is correct.

Proof of Theorem 5 Let V be the set of seeds used to construct possible region $P_G(V)$ in Step 2 of Algorithm 3. If $O_j \notin C_G$, the UV-edge $E_G(j)$ does not cut $P_G(V)$. In other words, $P_G(V) = P_G(V) - X_G(j)$. The I-pruning and C-pruning methods used in Step 2 also guarantee that c_j is not inside $P_G(V)$, that is, $c_j \notin P_G(V)$. Moreover, $u_i \subseteq u_G$. By substituting $S = V$ and $k = G$, we can deduce from Lemma 6 that $P_i(V) = P_i(V) - X_i(j)$. Now there are two cases to consider:

Case 1: O_j contributes an edge to $P_i(V)$. In other words, $O_j \in V$. Since an object in V is not pruned by Step 2 of Algorithm 3, $V \subseteq C_G$, and so $O_j \in C_G$. However, this contradicts with the assumption that $O_j \notin C_G$, and so this case cannot occur.

Case 2: O_j does not contribute an edge to $P_i(V)$. Since the UV-cell U_i of O_i must be inside $P_i(V)$, O_j cannot contribute an edge to U_i. Hence, O_j is not an r-object of O_i, and the theorem holds.

6 The UV-index

We now present the *UV-index*, an approximate version of the UV-diagram. The UV-index can be efficiently computed and stored. It also facilitates efficient query evaluation. Section 6.1 gives an overview of its structure. In Sect. 6.2, we discuss how to use this index to support execution of different queries. We explain its construction process in Sect. 6.3.

6.1 Structure of the UV-index

The UV-index adopts a framework similar to a quad-tree [5], in order to index the irregular and non-overlapping UV-partitions. Figure 11a illustrates this index.[4] Each non-leaf node, 16 bytes each, records a pointer to each of its four child nodes, where the square region spanned by each child node is one-fourth of that of its parent. The region covered by the root node is the whole domain D. Each leaf node stores all the objects whose UV-cells overlap with the region defined for the node. To save space, a node's region is not stored, since we can easily derive the dimension of the region based on the level of the node in the tree. Also, due to approximation, a

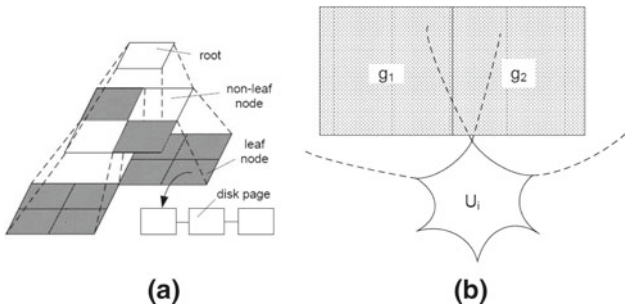


Fig. 11 UV-index: **a** structure, **b** overlap checking

UV-cell that does not overlap with the leaf node's region may be included. However, a UV-cell that truly overlaps with the region will not be excluded. For each leaf node l, we store a linked list of disk pages, which contain tuples <*ID, MBC, pointer*>, where:

- *ID* is the identity of object O_i whose UV-cell may overlap with the region covered by l;
- *MBC* is the circle that minimally bounds the uncertainty region of O_i; and
- *pointer* stores the disk page address of the object.

We assume that all non-leaf nodes are stored in the main memory, and allocate a maximum number of M non-leaf nodes. The leaf nodes, which contain the lists of pages, are stored in the disk. Hence, M controls the amount of main memory to be used to implement the index. Next, we study how to use it to support query evaluation.

6.2 Using the UV-index

We now explain how to use the UV-index to support the queries that we described in Sect. 3.2.

1. The PNN Query. To find the probabilistic nearest neighbors of q, we first locate the leaf node l, whose region contains q. This can be done easily by finding the grid that contains q in each index level, and traversing the index. We then retrieve the disk pages associated with l, which contains the *ID* and the *MBC* values of the objects stored in these pages. Since these objects may have their UV-cells overlap with the region of l, it is possible that q is contained in their UV-cells. Let L be the set of objects associated with l, and A be the answer objects of q. To answer a PNN, we need to retrieve A from L, where $A \subseteq L$. We use the method described in [15]: from the set of the *MBC*'s of the objects in L, find d_{minmax}, the minimum of the maximum distances of these objects from q. Any object with the minimum distance larger than d_{minmax} is removed, since it cannot have a nonzero qualification probability. For objects that are not filtered, their probabilities are computed and returned as answers.

[4] We adopt the quad-tree rather than the R-tree. While R-tree MBRs may overlap, quad-tree grids do not. Issuing a point query on non-overlapping UV-partitions in quad-tree is thus more convenient than R-tree.

2. The CPNN Query maintains the PNN answers for a "moving" query point, whose location is periodically reported to the server. Let q_0 be the latest position of q received by the server. Let g_0 be a leaf node in the UV-index, whose region r_0 contains q_0. We assume that the objects stored in the disk pages associated with g_0 are known. Now, suppose the new location of q, say, q_1, is received by the server. A straightforward solution is to treat q_1 as a new PNN query, and use the PNN algorithm described above to compute the answers of q_1. A better way is to check whether q_1 is inside r_0. If this is true, we simply use the object set associated with g_0 to compute the answer for q_1. This saves the effort of traversing the UV-index for q_1.

3. The UV-Partition Query. We append a counter to each leaf node, and record the number of objects at that node. This process could be done after the UV-index is constructed. Then, a range query with range R is issued over the index, in order to find the leaf nodes whose regions overlap with R. For every leaf node whose region r overlaps with R, we compute its density, which is equal to the number of objects associated with r, divided by the area of r. The query then outputs r and its density value.

4. The UV-cell Query. Notice that if an object O_i appears in a leaf node g, its UV-cell overlaps with the region of g. Hence, we can return the approximate area and the extent of O_i's UV-cell by scanning the leaf nodes associated with O_i, and then summing up the total area of the regions covered by these nodes. This step can be improved by precomputing and storing the area information. For example, we can scan all the leaf-nodes once, and generate a table for each O_i with its respective areas. A similar procedure can be used to support the operation of displaying the approximate shape of the UV-cell on the user's screen.

6.3 Construction of the UV-index

As discussed in Sect. 5, a UV-cell can be represented by a set of cr-objects, C_i. We now examine how this facilitates the construction of the UV-index.

Framework. Let g be the grid node being examined, and h_k (where $k = 1, \ldots, 4$) be the four child nodes of g. We define a variable *nonleafnum*, which indicates the number of non-leaf nodes allocated to the index and has an initial value of 1. Originally, the `root` of the grid is a leaf node, whose region covered (`root`.*region*) is the domain D.

We use Algorithm 4 (`InsertObj`) to insert an object O_i to the index. This algorithm, whose inputs are C_i and node g, is a recursive procedure, where `InsertObj` (`C`$_i$, `root`) is first invoked. In Step 1, `CheckOverlap` investigates whether the UV-cell represented by C_i overlaps with the region of grid g. If so, we check whether g is a non-leaf node. If this is true, `InsertObj` is called recursively (Steps 2–4). Otherwise, we perform `CheckSplit` (Step 7), which returns:

Algorithm 4 InsertObj

```
   Input: cr-objects C_i; Node g;
 1: if (CheckOverlap(C_i, g.region) = true) then
 2:     if g is a non-leaf node then
 3:         for k = 1 to 4 do
 4:             InsertObj(C_i, h_k);
 5:         end for
 6:     else
 7:         state ← CheckSplit(C_i, g);
 8:         switch (state)
 9:         case NORMAL:
10:             g.list.add(i, MBC(O_i), ptr(O_i));
11:         break;
12:         case OVERFLOW:
13:             Allocate new page for g;
14:             g.list.add(i, MBC(O_i), ptr(O_i));
15:         break;
16:         case SPLIT:
17:             delete g.list;
18:             for k = 1 to 4 do
19:                 Assign h_k as child of g;
20:             end for
21:             nonleafnum ← nonleafnum + 1;
22:         break;
23:     end if
24: end if
```

1. `NORMAL` (Steps 9–11): g's pages still have space left, and so $(i, MBC_i, ptr(O_i))$ is inserted to g's page, where $ptr(O_i)$ is the pointer to O_i's uncertainty region and pdf.
2. `OVERFLOW` (Steps 12–15): g's pages are full, and a new disk page has to be associated with g, before the information about O_i is inserted to the new page.
3. `SPLIT` (Steps 16–22): g's pages are full. The page list g is removed. Then, g becomes the parent of four nodes (h_k), which have been previously generated by `CheckSplit`. The region of each child node h_k covers each of the four quarters of the region defined for g. Also, *nonleafnum* is incremented by a value of 1. Essentially, The information about the UV-cells previously associated with g are now represented by its child nodes, and g becomes a non-leaf node.

Decision on Splitting. When g's pages are full, either O_i's information is inserted to a new page (`OVERFLOW`) or split into four child nodes (`SPLIT`). Ideally, the region of the leaf node that covers q is completely covered by a true UV-partition. This guarantees that the set of objects returned by the UV-index is the true answer objects. The UV-index, which contains grids, is just an approximation of the UV-diagram. Apparently, the more the splitting is performed, the closer the index can resemble the actual UV-diagram, and yield better query performance.

In fact, splitting is not always useful. Suppose that $g.region$ is associated with 100 UV-cells. Moreover, $g.region$ is *completely* covered by each of these UV-cells. Then, it is not necessary to redistribute g into four child nodes. If splitting is performed in this case, then the UV-cells associated with each child node are exactly the same.

Thus, more space is wasted to store duplicated information about the UV-cells. This can happen if the corresponding 100 objects of these UV-cells are close to each other. Then, these UV-cells have similar shapes and significant overlapping. To decide whether to split, we define *split fraction*, θ, as follows:

$$\theta = \frac{\min_{k=1,\ldots,4} |h_k.list|}{|g.list|} \tag{21}$$

which is the minimum fraction of UV-cells in one of the child nodes h_k that are also in g (note that the UV-cells associated with h_k must be the subset of the ones attached to g). A small θ means that the number of UV-cells overlapping with $h_k.region$ is small compared with that of g. We now define a splitting condition of a node:

Split if $\theta < T_\theta$

where $T_\theta \in [0, 1]$ is called the *split threshold*. A larger value of T_θ implies a higher tendency of splitting.

Algorithm 5 (`CheckSplit`) implements these ideas. Steps 1–3 return `NORMAL` if the pages of g are not full. Steps 4 and 5 return `OVERFLOW` if the number of non-leaf nodes allocated is higher than M. In Steps 7–16, we compute the value of θ, by creating four nodes h_k (Step 7), and checking the overlap of each UV-cell with $h_k.region$ (Steps 11 and 12). If the splitting condition is satisfied (Step 17), then the `SPLIT` decision is returned, where Algorithm 4 (Steps 18 and 19) will assign the nodes h_k to be the child nodes of g. Otherwise, the child nodes are deleted and an `OVERFLOW` decision is made (Steps 20 and 21).

Overlap Checking. Algorithm 6 tests whether the UV-cell of an object O_i overlaps with a grid g's region r. For every cr-object $O_k \in C_i$, if any of their corresponding outside region ($X_i(k)$) totally contains r, then `CheckOverlap` returns `false` (Steps 1–3). Otherwise, `true` is returned (Step 6). To prove the correctness, we use the following lemma:

Lemma 7 *If region r is totally covered by $X_i(k)$, where $O_k \in C_i$, then r must not overlap with the UV-cell U_i.*

Proof We would like to show that if there exists an object O_k, such that $r \subseteq X_i(k)$, then $r \cap U_i = \phi$. Let $\overline{X_i(j)}$ be the region $D - X_i(j)$. Then U_i, the UV-cell of O_i, can be expressed as the intersection of all regions $\overline{X_i(j)}$, for all objects in O except O_i, that is,

$$U_i = \bigcap_{j=1\ldots|O|\wedge j\neq i} \overline{X_i(j)} \tag{22}$$

Algorithm 5 CheckSplit

Input: cr-objects C_i; node g;
Outputs: `NORMAL, SPLIT, OVERFLOW`;
1: **if** there is space on any disk page of g.list **then**
2: return `NORMAL`;
3: **end if**
4: **if** $nonleafnum + 1 > M$ **then**
5: return `OVERFLOW`;
6: **else**
7: Create nodes h_k ($k = 1, \ldots, 4$) with $h_k.region$ equal to each quarter of $g.region$;
8: $A \leftarrow O_i \cup g.list$;
9: **for each** $O_j \in A$ **do**
10: **for each** h_k **do**
11: **if** (`CheckOverlap`(C_j, $h_k.region$)) = true **then**
12: h_k.list.add(j, $MBC(O_j)$, $ptr(O_j)$);
13: **end if**
14: **end for**
15: **end for**
16: $\theta \leftarrow (\min_{k=1,\ldots,4} |h_k.list|)/|g.list|$;
17: **if** $\theta < T_\theta$ **then**
18: return `SPLIT`;
19: **else**
20: delete h_k, where $k = 1, \ldots, 4$;
21: return `OVERFLOW`;
22: **end if**
23: **end if**

Algorithm 6 CheckOverlap

Input: cr-objects C_i; Region r;
Output: true if U_i and r overlap, false otherwise;
1: **for each** $O_k \in C_i$ **do**
2: **if** $r \subseteq X_i(k)$ **then** // Use 4-point testing
3: return `false`;
4: **end if**
5: **end for**
6: return `true`;

Since $r \subseteq X_i(k)$, we have

$$\begin{aligned} & r \cap \overline{X_i(k)} = \phi \\ \Rightarrow\ & (r \cap \overline{X_i(k)}) \cap \bigcap_{j=1\ldots|O|\wedge j\neq i\wedge j\neq k} \overline{X_i(j)} = \phi \\ \Rightarrow\ & r \cap (\overline{X_i(k)} \cap \bigcap_{j=1\ldots|O|\wedge j\neq i\wedge j\neq k} \overline{X_i(j)}) = \phi \\ \Rightarrow\ & r \cap U_i = \phi \end{aligned}$$

from Eq. 22. Hence, the lemma is correct.

To check whether a region r is contained in $X_i(j)$ (Step 2), a simple way is to generate and test with the UV-edge $E_i(j)$. This can be avoided, by carrying out a simple *4-point test*. Observe that r is a square, and the UV-edge of O_i with respect to O_j is concave in shape. If all its four corner points are confirmed to be in $X_i(j)$, we conclude that $r \subseteq X_i(j)$. Figure 11b shows that the region of g_1 must not overlap with U_i, since all the four corners of g are located on the outside region of one of the UV-edges. Checking whether a point is in $X_i(j)$ is easy, because we can simply check whether the point's minimum distance from O_i is larger than

its maximum distance from O_j. We thus use the four-point test in Step 2.

Notice that Algorithm 6 may incorrectly judge that U_i overlaps with r. Figure 11(b) shows that U_i does not overlap with the region of grid g_2. However, some corners of $g_2.region$ are not contained in the outside regions of two of the UV-edges of U_i. If this is true for *all* UV-edges of U_i, then U_i would be decided to be associated with g_2! If this happens, then during query evaluation, O_i will be retrieved from g_2. This increases the execution time since O_i is not in g_2. However, query accuracy is not affected, since we can still detect that O_i is not a nearest neighbor of q. In our experiments, this situation is rare, and does not have a significant effect on query evaluation time.

Since $|C_i| = O(n)$, Algorithm 6 needs $O(n)$ time to complete. Algorithm 5 uses $O(n^2)$ time, mainly for performing splitting and overlap checking with four child nodes. For Algorithm 4, each UV-cell, in the worst case, needs to perform overlap and split tests with M non-leaf nodes. Hence, its total time complexity is $O(Mn^2)$. The index has a maximum height of $M/4$, if, the data distribution is very skewed, and splitting always happens in one single quadrant. However, all non-leaf nodes, 16-byte long, can all be stored in the main memory. Thus, the tree height has little effect on query performance.

7 Results

We now report the results. Section 7.1 describes the experiment settings. In Sect. 7.2, we discuss the results about query performance. Section 7.4 presents the results about UV-index construction.

7.1 Setup

We use both synthetic and real datasets in our experiments. For synthetic data, we use Theodoridis et al's data generator[5] to obtain 20, 40, 60, 80, and 100K objects, which are uniformly distributed in a 10K × 10K space. Each object has a circular uncertainty region with a diameter of 40 units, and a Gaussian uncertainty pdf. For each uncertainty pdf, its mean is the center of the circle, and its variance is the square of one-sixth of the uncertainty region's diameter. We represent an uncertainty pdf as 16 histogram bars, where a histogram bar records the probability that the object is in that area. We also use three real datasets of geographical objects in Germany, namely *utility*, *roads*, and *rrlines*, with respective sizes 17, 30, and 36K. We also test the *Long Beach* (or *LB*) dataset, which contains 53K objects.[6] These objects are represented as circles before indexing, and has the same uncertainty pdf information as that of the synthetic data.

To compare with R-tree, we use a packed R*-tree [30] to index uncertain objects. The R-tree uses 4K-byte disk pages, and has a fanout of 100. We keep all its non-leaf nodes in the main memory.

For the UV-index, each non-leaf node has four 4-byte pointers to its children. We set M, the number of non-leaf nodes in the main memory, to be 10,000. The leaf nodes of both indexes, as well as the uncertainty information about the objects, are stored in the disk.

For T_θ, the splitting threshold used in constructing the UV-index, we have performed a sensitivity test. Under a wide range of T_θ, the indexes only have a slight performance difference. For very small values of T_θ (e.g., 0.2), however, the adaptive grid tends not to split, and degrades into long linked lists of pages. Here, we set T_θ to be 1.We wrote the programs in C++ and tested them on a Core2 Duo 2.66 GHz PC.

7.2 Results on query evaluation

We first study the performance of the queries studied in Sect. 3.2. We assume that the *LP* algorithm, presented in Sect. 5.4, is used to generate the UV-index. However, as we will discuss later, the different UV-index construction methods described here has little effect on query performance.

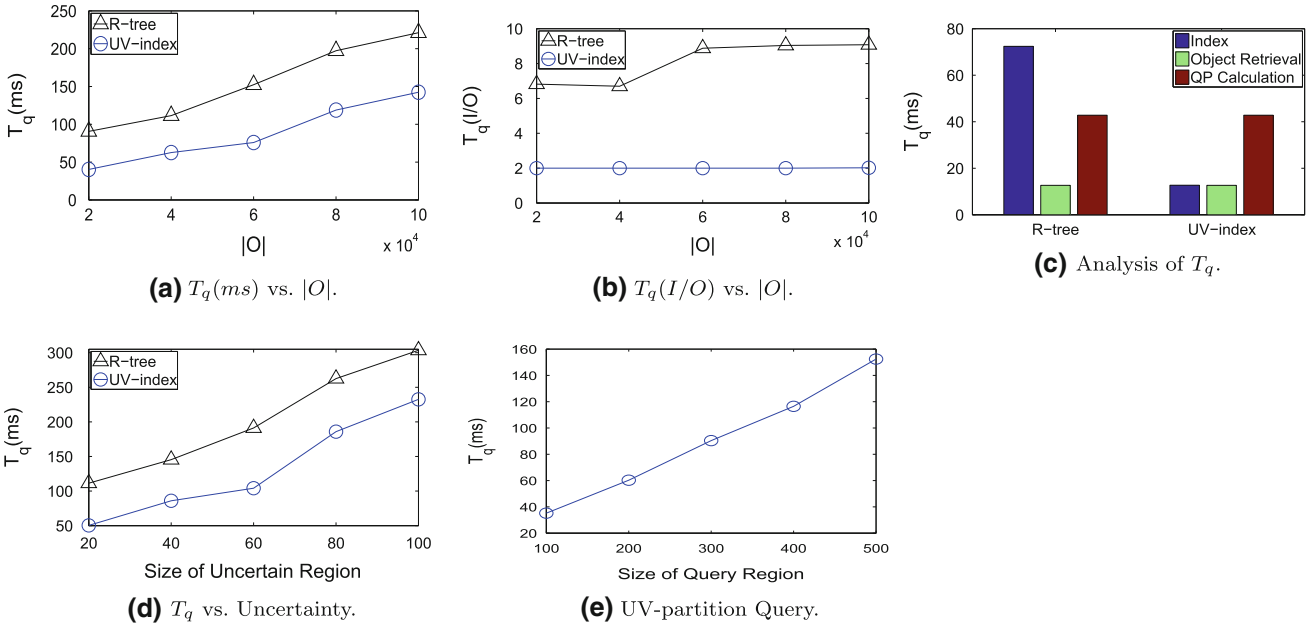
1. The PNN Query. We first compare the PNN performance of the UV-index and the R-tree. We present the average results of 50 query points randomly selected in the data domain. We use the numerical integration method of [15] to implement the probability computation of answer objects.[7] Figure 12a shows the query running time (T_q) for different synthetic datasets, with the number of objects ranging from 20 to 100K. The running times of both queries increase, because with a larger dataset, potentially more objects qualify as query answers, and this increases the time for index retrieval and probability computation. Our method outperforms R-tree in all cases. For example, when $|O| = 60$K, the UV-index needs about 50 % of the time needed by the R-tree.

To understand why our method performs better, let us examine the traversal time of the UV-index, which is composed of the time costs for visiting non-leaf and leaf nodes. Since its non-leaf traversal time takes little time in all experiments (up to 3.9 μs), we only present the I/O overhead. In Fig. 12b, we compare the I/O performance of the UV-index and the R-tree. The UV-index requires significantly less number of I/Os than the R-tree (e.g., when $|O| = 60$K, the UV-index consumes about one-fifth of the I/Os needed by

[5] http://www.rtreeportal.org/software/SpatialDataGenerator.zip.

[6] http://www.rtreeportal.org/.

[7] If faster methods such as [13] are used, the fraction of the time spent on retrieving answer objects from the index will be higher, and thus it would be more important to optimize the index.



(a) $T_q(ms)$ vs. $|O|$. **(b)** $T_q(I/O)$ vs. $|O|$. **(c)** Analysis of T_q.

(d) T_q vs. Uncertainty. **(e)** UV-partition Query.

Fig. 12 Results on the PNN query

Table 2 Results on real datasets

Dataset	$\|O\|$ (K)	T_q (UV) (ms)	T_q (R-tree) (ms)	T_c (s)	p_c (%)
utility	17	89	141	569	97.45
roads	30	82	135	1,195	97.80
rrlines	36	107	159	1,340	98.30
LB	53	109	173	1,579	98.22

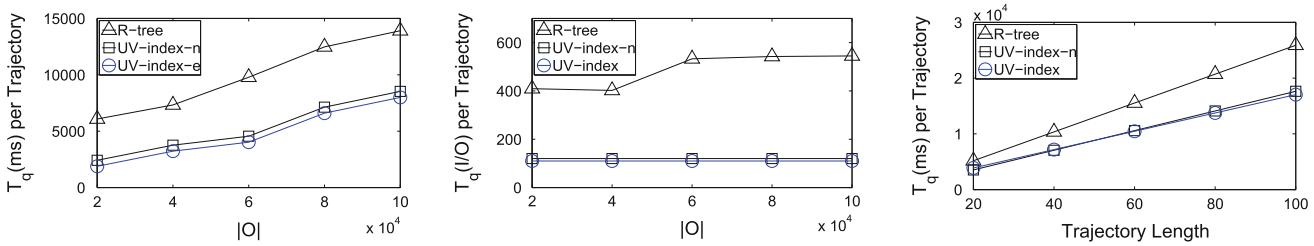
the R-tree). When the R-tree is used to process a PNN query, plenty of leaf nodes needed to be retrieved. For the UV-index, we only need to look for the leaf node that contains the query point. Since the number of disk pages for each leaf node is also small, a high I/O performance can be attained. Also notice that the number of I/Os for the R-tree increases with $|O|$, whereas that of the UV-index is relatively stable.

Figure 12c shows the time components of T_q: (1) index traversal; (2) retrieval of objects' pdf; and (3) probability computation. While object retrieval and probability computation costs are similar for both indexes, the R-tree requires a higher index traversal time. This explains the difference in Fig. 12b. In Fig. 12d, we can see that the query time of both indexes increases with uncertainty region size (i.e., the radius of the uncertainty region), since the larger the region, the more probable that the corresponding object is a PNN answer. For real datasets, columns 3 and 4 of Table 2 show that the UV-index consistently attains a higher query performance than the R-tree. Again, this is because the I/O performance of the UV-index is better than that of the R-tree.

2. The UV-Partition and the UV-cell Queries. We now examine the efficiency of our index for answering the UV-partition query on our synthetic dataset. For each size of a query region R, 50 queries are generated, whose centers of R are uniformly distributed in the data domain. We can see from Fig. 12e that the retrieval time of UV-partitions (T_q) increases with the size of R, since more UV-partitions are loaded when R becomes larger. The increase is almost linear, and the query evaluation time is less than 160 ms. We have also examined the performance of the UV-cell queries on the default synthetic dataset. On average, the time for obtaining a UV-cell from the UV-index is 58.46 ms, or equivalently, 4.62 I/Os. Thus, running a UV-cell query costs little time in our experiments.

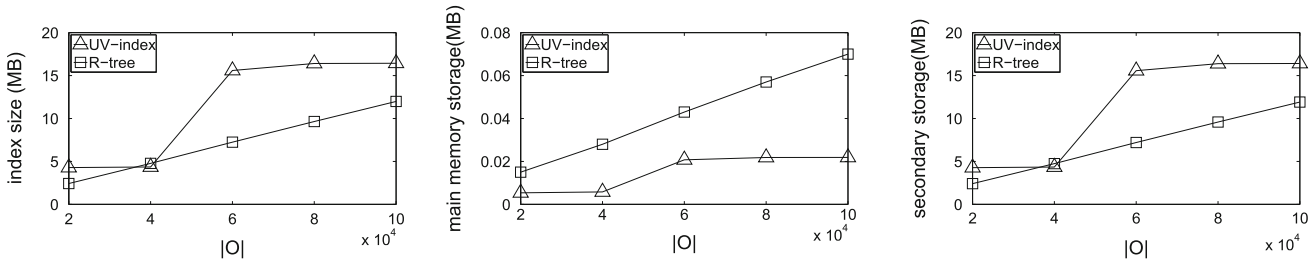
3. The CPNN query. To generate a CPNN query, we use the CanuMobiSim simulator,[8] which produces a moving-point trajectory. The movement of a query point follows a random walk model, as detailed in [34]. The location of a query point, which changes at a maximum speed of 100 units per second, is reported every second. The default "trajectory length" of a query is 60, that is, each query has 60 location

[8] http://canu.informatik.uni-stuttgart.de/mobisim/downloads/.



(a) $T_q(ms)$ vs. $|O|$. **(b)** $T_q(I/O)$ vs. $|O|$. **(c)** $T_q(ms)$ vs. Trajectory Length.

Fig. 13 Results on the CPNN query



(a) Index size vs. $|O|$· **(b)** main memory size vs. $|O|$. **(c)** disk size vs. $|O|$.

Fig. 14 Storage cost analysis

reports. In our experiments, each data point is the average of 50 queries.

We examine two algorithms that use the UV-index to support CPNN queries. The first variant, called *UV-index-n*, is a naïve application of the UV-index: each time a query point is received, the UV-index is consulted once. The second one, called *UV-index-e*, is the enhanced version of *UV-index-n*, where the UV-index is only consulted if the current query point is not located in the same grid as the previous one (Sect. 6.2). Figure 13a shows the evaluation time of a query over synthetic data of different sizes. As we can see, the query performance of the UV-index is at least 25 % times better than the R-tree. The reason can be explained by Fig. 13b, which shows the number of I/Os required by these methods. We observe that the I/O cost of issuing a CPNN on the UV-index is much lower than that of the R-tree. For example, when $|O| = 60k$, the query cost of the UV-index algorithms is about 30 % of the R-tree. We also see that *UV-index-e* performs better than *UV-index-n*. When the current query point q_1 is located in the grid g that also contains the previous query point q_0, *UV-index-e* uses the objects associated with g to answer the PNN at q_1. Thus, the effort of traversing the UV-index for q_1 can be saved. This saving is quite significant; at $|O| = 60k$, for instance, the number of I/Os required by *UV-index-e* is only 66 % of that of *UV-index-n*. In Fig. 13c, we examine the effect of the query trajectory length. Again, the *UV-index-e* performs the best among the three access methods.

7.3 Storage cost analysis

Next, we compare the sizes of R-tree and UV-index. As mentioned in Sect. 7.1, for both indices, we store the non-leaf nodes in the main memory, and the leaf-nodes in the disk. The index size is the sum of the main memory and disk space required. Figure 14a compares the size of the UV-index and the R-tree. The UV-index is larger than the R-tree. While the UV-index consumes less main memory than the R-tree (Fig. 14b), it needs more disk space (Fig. 14c). Although the UV-index has a larger size than the R-tree, the UV-index provides a better query performance. Moreover, the UV-index provides functionalities that are not available by R-tree (e.g., retrieval of UV-partitions). These benefits are provided in the expense of a larger disk cost. Given the low cost of hard disk space nowadays, we believe that the extra disk space required by the UV-index is still justifiable.

7.4 Results on UV-Index Construction

We now examine several UV-index construction methods. We first study the following techniques:

- *Basic*: a UV-cell is derived using Algorithm 1, which is then used to build the UV-index;
- *ICR (I- and C-pruning with Refinement)*: collect cr-objects through I- and C-pruning (Algorithm 2), com-

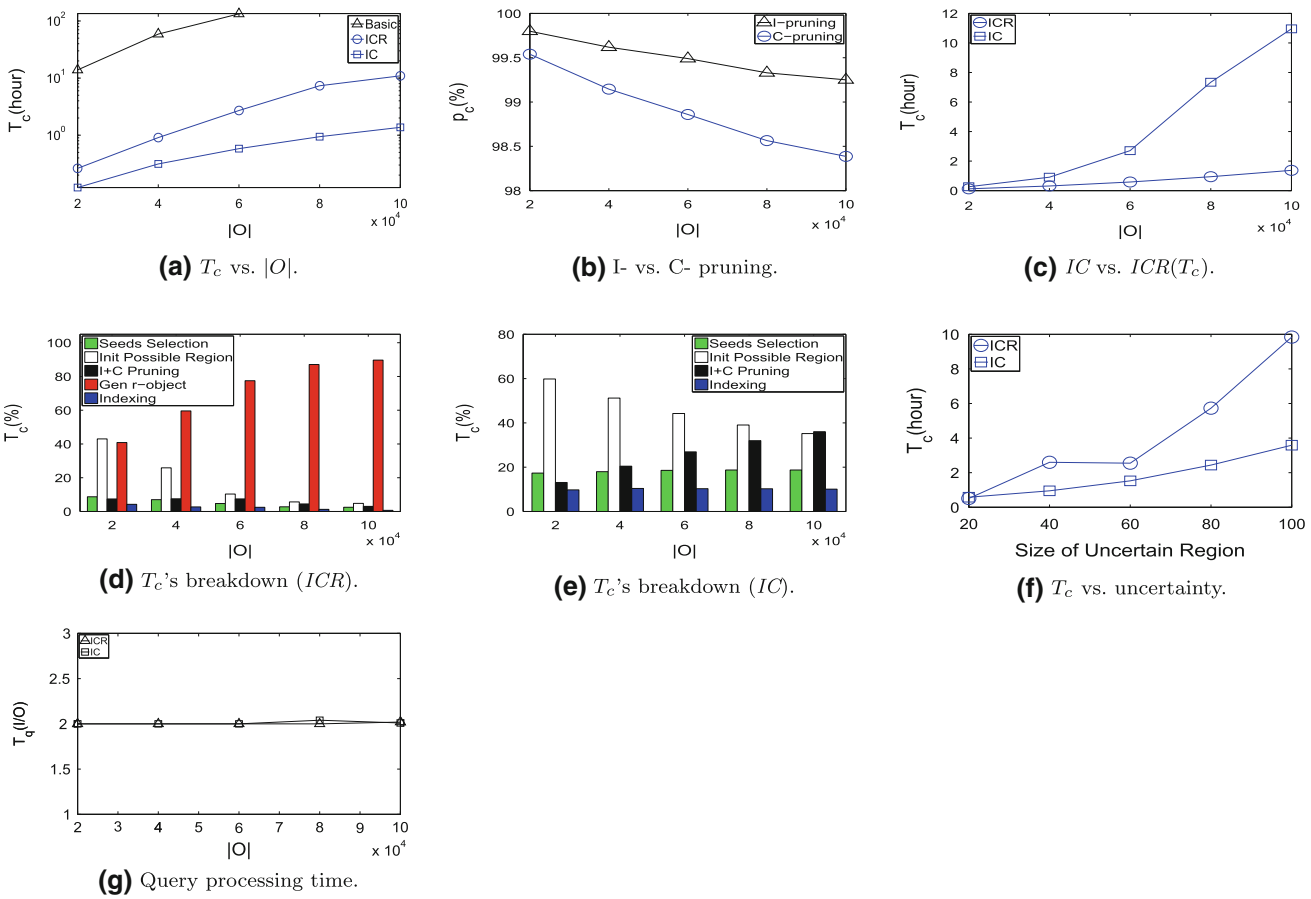


Fig. 15 *Basic*, *ICR*, and *IC*

pute UV-cells and obtain the r-objects, then index them with Algorithm 4.

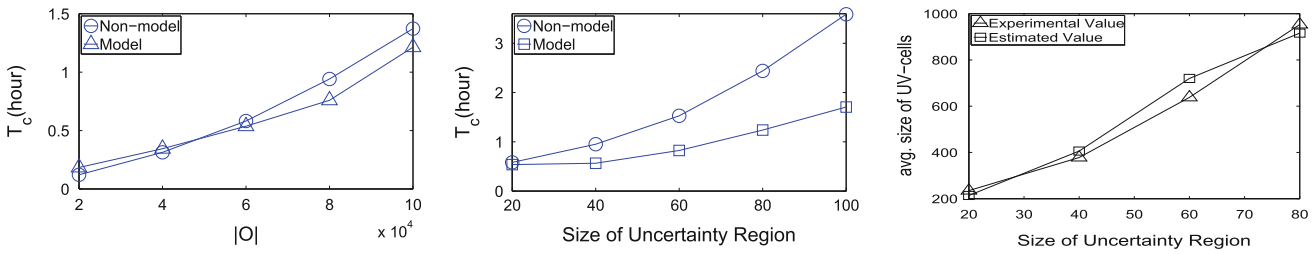
– *IC (I- and C-pruning)*: the cr-objects obtained through I- and C-pruning are used directly to construct the UV-index by Algorithm 4.

We assume that the R-tree for uncertain objects is available for use by these methods. Unless stated otherwise, the model-based seed selection and batch construction methods are not used (their effect will be examined later). For generating initial possible regions (used in *IC* and *ICR*), we set k to 300 for performing the k-NN search. Then, the domain D is divided into $k_s = 30$ sectors to obtain the seeds.

Figure 15a describes the development time (T_c) of the UV-index for the three methods. *Basic* increases sharply with the dataset size; handling a $40K$ dataset requires about 60 h. This is because constructing a UV-cell requires an exponential amount of time and numerous complex hyperbola intersections. For *ICR* and *IC*, the use of I- and C-pruning significantly reduces the number of objects examined. Their effects are shown in Figure 15(b), where p_c, the pruning ratio, denotes the fraction of objects from O that has been filtered. At $|O|$=60k, I-pruning and C-pruning achieve a pruning ratio of 98.9 and 99.5 % respectively. Hence, a large portion of objects are removed before being considered for constructing the UV-cell. Next, we examine *ICR* and *IC*.

IC versus ICR. As shown in Fig. 15c, *IC* performs much better than *ICR*. For example, at $|O| = 80$K, the construction time of *IC* is about 10 % of that of *ICR*. To understand why, we analyze their time components in Fig. 15d, e. Recall the difference between the two methods is that *ICR* needs to find out the exact r-objects (by constructing an exact UV-cell based on the objects returned by pruning), while *IC* does not. For *ICR*, Fig. 15d shows the fraction of the construction time spent on: (1) seeds selection, (2) initial possible region computation, (3) I- and C-pruning, (4) generating r-objects, and (5) indexing UV-cells. For most datasets, *ICR* spends most of the time to generate exact r-objects, which is very costly. For *IC*, r-object is not produced (Fig. 15e). Instead, the cr-objects produced by the pruning methods are immediately passed to Algorithm 4 for indexing. The number of cr-objects generated, while larger than that of r-objects, does not increase the indexing time significantly.

In Fig. 15f, the construction time of *ICR* increases sharply with the objects' uncertainty region sizes. With larger uncertainty regions, it is more likely that these regions overlap with



(a) T_c vs. $|O|$. **(b)** T_c vs. U. **(c)** S vs. U.

Fig. 16 Model-based seed selection

each other, making it harder to prune the objects, so that more time is needed to generate r-objects. On the other hand, *IC* is relatively insensitive to the change in uncertainty region sizes.

We have also measured the query times between the indexes created by *IC* and *ICR*. Figure 15g shows that the UV-index generated by the two methods is highly similar, resulting in a close query performance. The query cost of *ICR* is about 0.01 I/Os, or 0.13 ms, better than *IC*. In the sequel, we assume that *IC* is used.

Model-based index construction. In Sect. 5.2, we have demonstrated how to use the UV-cell model (Sect. 4.4) to facilitate seed selection for objects whose locations are uniformly distributed. We call the UV-index construction algorithm that employs this method as *Model*, and the one that does not use it as *Non-model*. We evaluate these two algorithms on our synthetic datasets. As we can see from Fig. 16a, *Model* performs better than *Non-model* in most cases. When $|O|$=80 k, about 20 % of the index construction time is saved. Figure 16b illustrates that *Model* is consistently better than *Non-model* under different uncertainty region sizes. For example, when the radius of an uncertainty region is 80, the time required by *Model* is about half of that of IC.

To understand why *Model* performs well, we compare the difference between the size S of a UV-cell estimated by our model, and its "true" size. Again, S is the length of the MBR that tightly bounds the estimated UV-cell. The true size of the UV-cell can be obtained by using Algorithm 1. Based on the vertices of this UV-cell, we obtain its minimum bounding rectangle (MBR). We use the larger length of the two dimensions of this MBR to represent the size of the UV-cell. Figure 16c shows the average size of a UV-cell under different uncertainty region sizes. The UV-cell size increases with the uncertainty region radius, since an object can be in more possible locations. This increases its chance to be a possible nearest neighbor of a query point. In this experiment, our method offers a reasonable estimation of the UV-cell's size—the estimation error is between 4 and 12 %. This enables the selection of seeds, as well as the index construction algorithm, to be effective.

Batch processing. We next examine the performance of *LP*, which derives cr-object based on groups of data objects (Sect. 5.4). We compare *LP* with *single*, which generates a cr-object set for each data object separately. We do not use model-based seed selection in these experiments. Figure 17a shows that *LP* performs better than *single* on our synthetic datasets. At $|O| = 80$k, the time cost of *LP* is about 60 % of that of *single*. In *LP*, the cr-object set generation cost is shared among a group of objects.

We also test the performance of *single* and *LP* on larger datasets. We use the same synthetic data generator to produce two datasets that contain 0.5M and 1M objects. The 1M dataset occupies 640Mbytes. The new result, illustrated in Fig. 17b, shows that the construction performance of both *single* and *LP* increases with the dataset size in a linear manner. For the 1M dataset, *LP* needs 7.7 h, which is 23 % faster than *single*.

Figure 17c shows that when *LP* is used, the seed selection time of *single* is shortened by more than 80 %. While *single* generates seeds for every object individually, in *LP*, the seeds of every object in set G are retrieved from a set of objects C_G (Step 2 of Algorithm 3). We can also see that the I- and C-pruning time required by *LP* is also less than *single*; when $|O| = 60k$, the improvement is over 60 %. In *single*, I-pruning is done for every object; in *LP*, I-pruning is only done once for every group. The performance gap is more profound when $|O|$ is large, since the same domain is populated with more objects, resulting in more candidates retrieved after I-pruning.

We also examine the effect of the average uncertainty region size on the construction time. As discussed before, the larger this size, the more construction time will be needed. Figure 17d shows that *LP* is more stable than *single*. When the uncertainty region size is 60, *LP* needs more about 60 % time of *single*; when the size becomes 100, *LP* is 3.5 times faster than *single*. In Fig. 17e, we compare the query performance of the UV-indices generated by *single* and *LP*. We observe that the number of I/Os required by the two methods is the same. Their probability computation times, not shown here, are also very close. Hence, the query performance of two methods is almost the same.

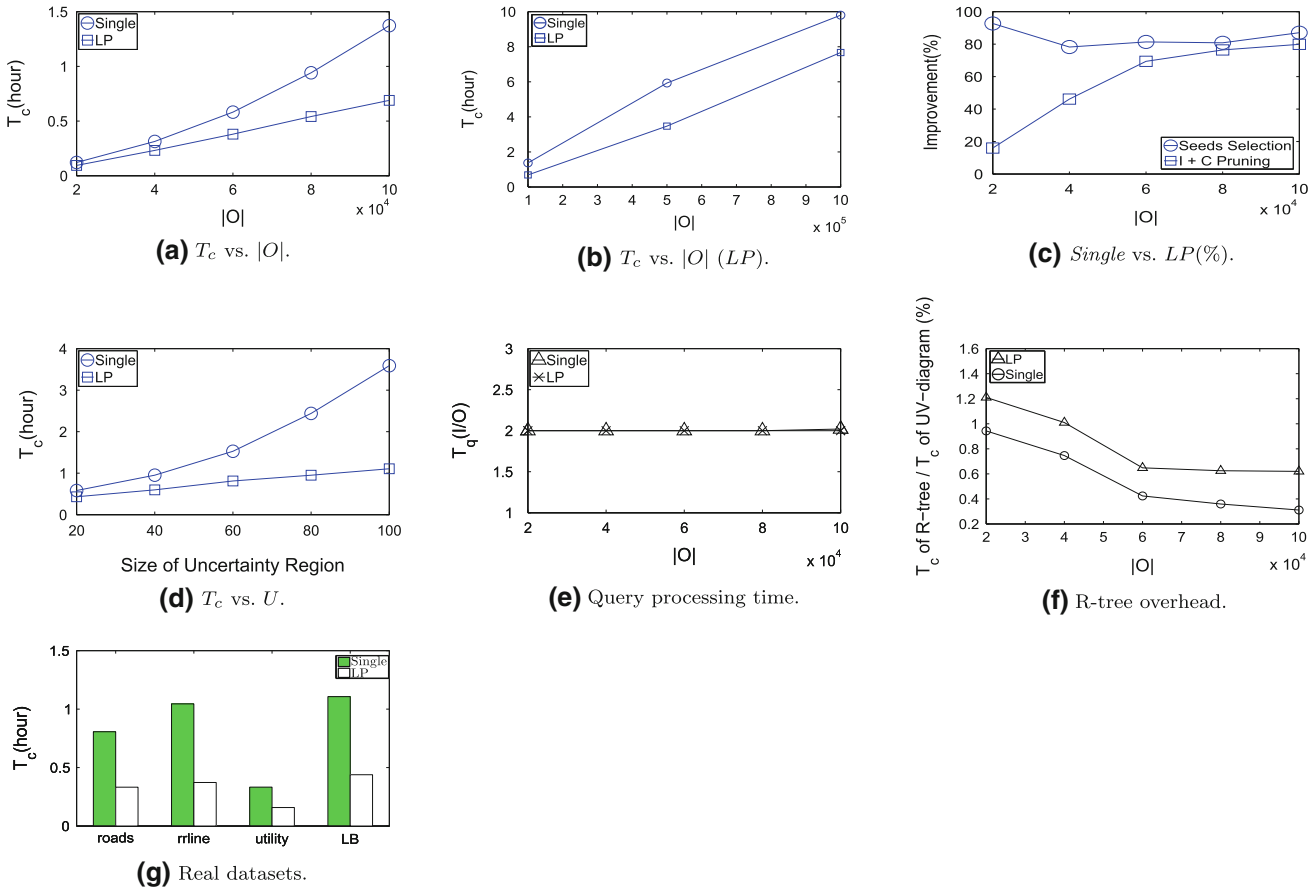


Fig. 17 Results on batch processing

Next, we compare the construction time of the R-tree and UV-index, using *single* and *LP*. Figure 17f shows that the construction cost of the R-tree is less than 1 % of that of the UV-index. Hence, the R-tree introduces little overhead to the UV-index construction process. However, it improves the performance of generating the UV-index. For instance, the I-pruning phase can be executed more efficiently with the use of the R-tree.

For real datasets, *LP* also outperforms *single* (Fig. 17g). In *rrline*, for example, *LP* needs one-third of the time required by *single*. *LP* also achieves a high pruning ratio, as shown in Table 2. This explains why *LP* requires less time to construct a UV-index, compared with *single*.

Skewness. In Fig. 18, we study the effect of data skewness, by varying the variance (σ) of the objects' mean positions. We can see that when the data are more skewed (i.e., with a smaller variance), the construction time is higher, because in a dense area where uncertainty regions have high degree of overlap, an object's UV-cell is likely small and associated with many r-objects. The *LP* algorithm is still more efficient than *single*. In the most skewed dataset that we tested (σ =1,500), *LP* is 33.3 % faster than *single*.

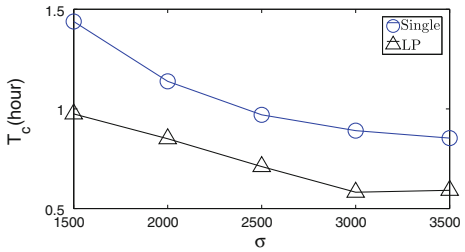


Fig. 18 Effect of variance

Finally, we examine how a skewed distribution of the centers of uncertainty regions can affect our results. We obtain a 60k dataset that follows the zipfian distribution, by using the same generator that produces our uniformly distributed dataset. For the zipfian distribution, the average query I/O costs for *IC* and *ICR* are 2.48 and 2.41. Thus, the query performance of ICR is 0.07 I/Os (or 2.8 %) better than IC. Since their time difference is small (around 0.4 ms), we use *IC* in the rest of the experiments.

Table 3 compares these two distributions in terms of their construction and query performance, by using the batch

Table 3 Results on zipfian distribution

$\|O\| = 60k$	Uniform	Zipfian	
	LP	*Single*	*LP*
T_c(hours)	0.45	5.78	2.46
$T_q(I/Os)$	2.00	2.48	2.45

processing (*LP*) technique. Observe that the construction time of the zipfian distribution is worse than the uniform distribution. In a skewed dataset, a UV-cell in a very dense area can be determined by many r-objects, and this renders lower pruning efficiency in the construction phase. However, there is only a slight query I/O difference between the two distributions, and the query performance for both distributions is almost the same.

In the same table, we study the difference between *single* and *LP* for zipfian distribution. Notice that *LP* requires about 42 % of time needed by *single*. This means that our batch processing method improves the construction performance for zipfian distribution significantly. The query performance of the UV-index constructed by *LP* is also slightly better (0.03 I/Os) than *single*.

8 Conclusions

The UV-diagram is a variant of the Voronoi diagram designed for uncertain data. To tackle the complexity of constructing and evaluating a UV-diagram, we introduce the concept of UV-cells and cr-objects. We study the theoretical size of a UV-cell. We propose an adaptive index for the UV-diagram, and develop efficient algorithms for building it. We also present a batch processing algorithm to further reduce the UV-index construction time. Our experiments show that this index efficiently supports PNNs and other UV-diagram-related queries.

We plan to study the use of the UV-diagram to support other variants of probabilistic NNQs, for example, approximate NNQs [12,13]; monochromatic and bichromatic reverse-nearest-neighbor (RNN) queries [10,27,42]; and k-RNN queries [11]. Another interesting problem is to design a UV-diagram such that whenever a query point is located in a UV-cell U_i, we can know that the qualification probability of O_i is larger than some threshold T. By using this variant of UV-diagram, we can get all the objects with qualification probability larger than T, without computing their actual probabilities. This could be beneficial to queries where a user is only interested in answers with qualification probabilities larger than T. It is also interesting to examine how the UV-diagram can support multi-dimensional data and incremental updates.

Acknowledgments Reynold Cheng, Xike Xie, Liwen Sun, and Jinchuan Chen were supported by the Research Grants Council of Hong Kong (GRF Projects 711110, 711309E, 513508). We would like to thank the anonymous reviewers for their insightful comments.

9 Appendix 1: Hyperbolic curve intersection

As discussed in Sect. 3.1, a vertex of the UV-cell is the intersection point of two hyperbolic curves. We now outline the procedure of finding this intersection, using the method described in [3]. We can represent two hyperbolic curves, C_1 and C_2, as homogeneous conic equations:

$$C_1: A_1x^2 + 2B_1xy + C_1y^2 + 2D_1xz + 2E_1yz + F_1z^2 = 0$$
$$C_2: A_2x^2 + 2B_2xy + C_2y^2 + 2D_2xz + 2E_2yz + F_2z^2 = 0$$

which are obtained by substituting x/z into x and y/z into y for the hyperbolas (Eq. 7) of C_1 and C_2. Next, we construct equation C_λ:

$$C_\lambda : C_1 + \lambda C_2 = 0 \tag{23}$$

where λ is a real value, and C_λ, a linear combination of C_1 and C_2, is a system of hyperbolas. We then rewrite C_λ in the form of $\omega^T H\omega = 0$, where $\omega = (x, y, z)^T$, and

$$H = \begin{pmatrix} A_1 + \lambda A_2 & B_1 + \lambda B_2 & D_1 + \lambda D_2 \\ B_1 + \lambda B_2 & C_1 + \lambda C_2 & E_1 + \lambda E_2 \\ D_1 + \lambda D_2 & E_1 + \lambda E_2 & F_1 + \lambda F_2 \end{pmatrix}$$

Let $det(H)$ be the determinant of H. Our aim is to find the value(s) of λ that satisfy the characteristic equation $det(H) = 0$. The real value of λ, when substituted into Eq. 23, ensures that (1) there is at least one intersection between C_1 and C_2, and (2) C_λ becomes a degenerated hyperbola, in the form of two straight lines.

Finally, for each of the λ found from the characteristic equation, we obtain at most four roots that simultaneously satisfy C_λ and C_1. Each root represents an intersection point of C_1 and C_2.

10 Appendix 2: Properties of a possible region (Lemma 1)

Given an object O_i, we discuss two properties of its possible region P_i.

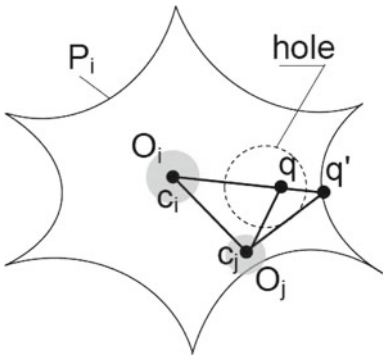


Fig. 19 Illustrating the proof of Lemma 1

1. Connectivity of P_i: Recall from Definition 3 that P_i is the intersection of a set of inside regions. Since each inside region is a connected region (by Definition 2), P_i must also be connected.

2. P_i **cannot contain any hole inside it.** Suppose by contradiction that there is a hole h inside P_i, such that an arbitrary point q inside h does not have O_i as its possible nearest neighbor. Figure 19 illustrates the situation. Since q must be covered by the UV-cell of some other object, let us assume that q is covered by the UV-cell of object O_j. Then, $dist_{min}(q, O_i) > dist_{max}(q, O_j)$, or $dist(q, c_i) - r_i > dist(q, c_j) + r_j$.

We now draw a straight line, which passes through c_i and q, and intersects the boundary of P_i at q'. We have:

$$\begin{aligned} &dist_{min}(q, O_i) + dist(q, q') > dist_{max}(q, O_j) \\ &+ dist(q, q') \\ &\Rightarrow dist(q, c_i) - r_i + dist(q, q') > dist(q, c_j) \\ &+ r_j + dist(q, q') \\ &\Rightarrow (dist(q, c_i) + dist(q, q')) - r_i > (dist(q, c_j) \\ &+ dist(q, q')) + r_j \end{aligned}$$

Since $dist(q, c_i) + dist(q, q') = dist(q', c_i)$ and $dist(q, c_j) + dist(q, q') > dist(q', c_j)$, we have:

$$dist(q', c_i) - r_i > dist(q', c_j) + r_j$$

In other words, $dist_{min}(q', O_i) > dist_{max}(q', O_j)$. Hence, q' cannot have O_i as its nearest neighbor. However, this is not possible, since $q' \in P_i$. Therefore, P_i cannot have any hole.

11 Appendix 3: Size of a possible region (Lemma 2)

Here, we explain how to derive the size of a possible region, as shown in Eq. 8, Sect. 4. Let us denote the six objects that have the same distance d from O_1 be $\{O_2, \ldots, O_7\}$, as shown in Fig. 6. We consider two UV-edges $E_1(2)$ and $E_1(3)$. Let X_0 be the intersection of $E_1(2)$ and $E_1(3)$. Using Eq. 1, we have:

$$\begin{cases} dist_{min}(O_1, X_0) = dist_{max}(O_2, X_0) \\ dist_{min}(O_1, X_0) = dist_{max}(O_3, X_0) \end{cases} \quad (24)$$

Let X_1 be the point on O_1 such that $dist(X_0, X_1) = dist_{min}(X_0, O_1)$. Also, let $X_2(X_3)$ be the point on $O_2(O_3)$ whose distance from X_0 is the maximum between X_0 and $O_2(O_3)$. According to Eq. 24,

$$dist(X_0, X_1) = dist(X_0, X_2) = dist(X_0, X_3) \quad (25)$$

Since X_1, X_2 and X_3 have the same distance to X_0, they are on a circle centered at X_0 with radius R. Thus, as shown in Fig. 6, X_0 is the center of circle $\odot(X_0, R)$, which is externally tangent to O_1 on X_1, and internally tangent to O_2 (O_3) on X_2 (X_3). Therefore,

$$\begin{cases} dist(X_0, c_1) = R + r \\ dist(X_0, c_2) = dist(X_0, c_3) = R - r \end{cases} \quad (26)$$

Now, let the coordinates of c_1 be $(c_1.x, c_1.y)$. Since $\angle c_2 c_1 X_0 = \frac{\pi}{6}$ (according to [40]), we have $c_2 = (c_1.x - \frac{d}{2}, c_1.y + \frac{\sqrt{3}d}{2})$, and $X_0 = (c_1.x, c_1.y + R + r)$. By substituting them to Eq. 26, we have:

$$R = \frac{d \times (d - \sqrt{3}r)}{\sqrt{3}d - 4r} \qquad (d > \frac{4r}{\sqrt{3}}) \quad (27)$$

Notice that d has to be larger than $\frac{4r}{\sqrt{3}}$, in order for R to be positive. The dimension of the square s that bounds the possible region $P_{1,d}$ is then equal to $s = 2 \times (R + r)$. By substituting R with Eq. 27, we can obtain Eq. 8.

12 Appendix 4: Size of a UV-cell (Theorem 1)

Here, we establish the condition that the possible region P_{1,d_0}, formed by the six objects in $H(d_0)$, is exactly the UV-cell of O_1. Recall that the centers of uncertain regions of objects in $H(d_0)$, which are the closest to that of O_1, form the vertices of a hexagon HEX_1, as shown in Fig. 7. Now, if objects in $H(d_0)$ are disregarded, then any of the object O_k whose uncertainty region center is a vertex of hexagon HEX_2 must be the nearest neighbor of O_1. Suppose that the UV-edge $E_i(k)$ cannot contribute to P_{1,d_0}. Then, as all uncertainty regions are equally spaced and identical, the UV-edges of other objects that are further away from HEX_1 and HEX_2 must also not change the shape of P_{1,d_0}. Thus, P_{1,d_0} becomes the UV-cell of O_1, i.e., U_1.

When does $E_i(k)$ fail to influence the shape of P_{1,d_0}? First, we calculate the minimum distance between the center of O_1 and $E_1(k)$, which is equal to $\frac{\sqrt{3}}{2}d_0 + r$. We compare this with $\frac{s(d_0)}{2}$, where $s(d_0)$ is the size of the square that bounds

P_{1,d_0} according to Eq. 8. If $\frac{s(d_0)}{2} < \frac{\sqrt{3}}{2}d_0 + r$, U_1, which is embedded in the square of size $s(d_0)$, cannot be further refined by $E_1(k)$. By substituting this condition into Eq. 8, we have:

$$\frac{s(d_0)}{2} < \frac{\sqrt{3}}{2}d_0 + r$$
$$\Rightarrow \frac{1}{2} \times \frac{2d_0^2 - 8r^2}{\sqrt{3}d_0 - 4r} < \frac{\sqrt{3}}{2}d_0 + r$$

Assume that $d_0 > \frac{4r}{\sqrt{3}}$ (required by Lemma 2). By multiplying $2(\sqrt{3}d_0 - 4r)$ on both sides of the above inequality, we have:

$$2d_0^2 - 8r^2 < (\sqrt{3}d_0 + 2r)(\sqrt{3}d_0 - 4r)$$
$$\Rightarrow 2d_0^2 - 8r^2 < 3d_0^2 + 2\sqrt{3}d_0r - 4\sqrt{3}d_0r - 8r^2$$
$$\Rightarrow 2d_0^2 < 3d_0^2 - 2\sqrt{3}d_0r$$
$$\Rightarrow d_0 > 2\sqrt{3}r$$

Thus, $d_0 > 2\sqrt{3}r$ is the condition that $E_i(k)$ cannot change P_{1,d_0}. It is also the constraint that the 6 objects of HEX_1 form the square of dimension $s(d_0)$ that minimally U_1.

13 Appendix 5: Proof of Lemmas for Section 5.5

13.1 Proof of Lemma 4

For any point $p \in P_i(S)$, p is within the intersection of the inside regions $\overline{X_i(j)}$, where $O_j \in S$. Hence, for every $O_j \in S$,

$$dist_{max}(p, O_j) > dist_{min}(p, O_i) \tag{28}$$

Since $u_i \subseteq P_i(S)$, and $u_i \subseteq u_k$, $\forall p \in P_i(S)$, $dist_{min}(p, O_i) \geq dist_{min}(p, O_k)$. Using Eq. 28, we have:

$$\forall p \in P_i(S), O_j \in S : dist_{max}(p, O_j) > dist_{min}(p, O_k) \tag{29}$$

This means that $\forall p \in P_i(S)$, $p \in \bigcap_{O_j \in S} \overline{X_k(j)}$, or simply $\forall p \in P_i(S)$, $p \in P_k(S)$. Thus, $P_i(S) \subseteq P_k(S)$, and the lemma is proved.

13.2 Proof of Lemma 5

Proof **(If)** Since $p \in P_i(S)$, we have $p \in \cap_{O_k \in S} \overline{X_i(k)}$. Thus, for every $O_k \in S$,

$$dist_{max}(p, O_k) > dist_{min}(p, O_i) \tag{30}$$

Using Eqs. 16 and 30, we see that $dist_{max}(p, O_j) > dist_{min}(p, O_i)$. So, the "if" part is proved.

(Only if) Consider any point p' lying on some UV-edge $E_i(k)$ of $P_i(S)$, where $O_k \in S$. Then,

$$dist_{max}(p', O_k) = dist_{min}(p', O_i) \tag{31}$$

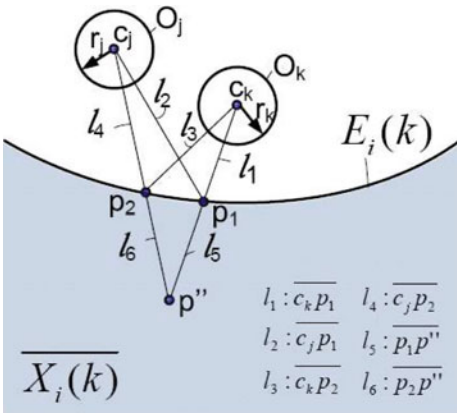


Fig. 20 Illustrating the proof of Lemma 5

Using Eqs. 31 and 15, we have:

$$dist_{max}(p', O_j) > dist_{max}(p', O_k) \tag{32}$$

Thus, the "only if" part is true for any $p' \in E_i(k)$.

We now complete the proof by showing that the lemma is true for any $p'' \in P_i(S)$. Since $c_j \notin P_i(S)$, a line that passes through c_j and p'' must intersect $E_i(k)$ at some point p_2 for some object $O_k \in S$. Also suppose a line that passes through c_k and p'' intersects $E_i(k)$ at p_1. The situation is shown in Fig. 20.

Using Eq. 32, we have:

$$dist_{max}(p_2, O_j) > dist_{max}(p_2, O_k)$$

This implies:

$$l_4 + r_j > l_3 + r_k$$
$$l_4 + l_6 + r_j > l_3 + l_6 + r_k$$

Using triangular inequality, we have:

$$l_3 + l_6 > l_1 + l_5$$

Thus,

$$l_4 + l_6 + r_j > l_1 + l_5 + r_k$$

or simply $dist_{max}(p'', O_j) > dist_{max}(p'', O_k)$. Thus, the "only if" part is correct.

References

1. Aggarwal, C.C.: On unifying privacy and uncertain data models. In: ICDE (2008)
2. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: SIGMOD (2000)
3. Akopyan, A., Zaslavski, A.: Geometry of Conics. American Mathematical Society, Providence, RI (2007)
4. Albers, G., Mitchell, J.S., Guibas, L.J., Roos, T.: Voronoi diagrams of moving points. Intl. J. Comput. Geom. Appl. **8**(3), 365–380 (1998)

5. Aref, W., Ilyas, I.: Sp-gist: an extensible database index for supporting space partitioning trees. JIS. **17**(1), 215–290 (2001)
6. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD (1990)
7. Berchtold, S., Ertl, B., Keim, D.A., peter Kriegel, H., Seidl, T.: Fast nearest neighbor search in high-dimensional space. In: ICDE (1998)
8. Beskales, G., Soliman, M., Ilyas, I.: Efficient search for the top-k probable nearest neighbors in uncertain databases. In: VLDB (2008)
9. Chazelle, B., Edelsbrunner, H.: An improved algorithm for constructing kth-order voronoi diagrams. IEEE Trans. Comput. **36**(11), 1349–1354 (1987)
10. Cheema, M.A., Lin, X., Wang, W., Zhang, W., Pei, J.: Probabilistic reverse nearest neighbor queries on uncertain data. TKDE **16**(9), 550–564 (2009)
11. Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: Influence zone: Efficiently processing reverse k nearest neighbors queries. ICDE (2011)
12. Chen, J., Cheng, R., Mokbel, M., Chow, C.-Y.: Scalable processing of snapshot and continuous nearest-neighbor queries over one-dimensional uncertain data. VLDB J. **18**(5), 1219–1240 (2009)
13. Cheng, R., Chen, J., Mokbel, M., Chow, C.-Y.: Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In: ICDE (2008)
14. Cheng, R., Kalashnikov, D., Prabhakar, S.: Evaluating probabilistic queries over imprecise data. In: SIGMOD (2003)
15. Cheng, R., Kalashnikov, D., Prabhakar, S.: Querying imprecise data in moving object environments. TKDE **16**(9), 1112–1127 (2004)
16. Cheng, R., Xia, Y., Prabhakar, S., Shah, R., Vitter, J.S.: Efficient indexing methods for probabilistic threshold queries over uncertain data. In: VLDB (2004)
17. Cheng, R., Xie, X., Yiu, M.L., Chen, J., Sun, L.: UV-diagram: a voronoi diagram for uncertain data. In: ICDE (2010)
18. Dalvi, N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: VLDB (2004)
19. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf O.: Computational Geometry: Algorithms and Applications. Springer, New York (1997)
20. Hua, M., Pei, J., Zhang, W., Lin, X.: A probabilistic threshold approach. In: SIGMOD, ranking queries on uncertain data (2008)
21. Jooyandeh, M., Mohades, A., Mirzakhah, M.: Uncertain voronoi diagram. Inf. Process. Lett. **109**(13), 709–712 (2009)
22. Kao, B., Lee, S., Cheung, D., Ho, W., Chan, K.: Clustering uncertain data using voronoi diagrams. In: ICDM (2008)
23. Karavelas, M.I.: Voronoi diagrams for moving disks and applications. In: WADS (2001)
24. Kriegel, H., Kunath, P., Renz, M.: Probabilistic nearest-neighbor query on uncertain objects. In: DASFAA (2007)
25. Lian, X., Chen, L.: Monochromatic and bichromatic reverse skyline search over uncertain databases. In: SIGMOD (2008)
26. Lian, X., Chen, L.: Probabilistic group nearest neighbor queries in uncertain databases. TKDE **20**(6), 809–824 (2008)
27. Lian, X., Chen, L.: Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data. In: VLDBJ (2009)
28. Ljosa, V., Singh, A.: APLA: Indexing arbitrary probability distributions. In: ICDE (2007)
29. Ljosa V., Singh, A.: Top-k spatial joins of probabilistic objects. In: ICDE (2008)
30. Hadjieleftheriou, M.: Spatial index library version 0.44.2b
31. Mokbel, M., Chow, C., Aref, W.: The new casper: query processing for location services without compromising privacy. In: VLDB (2006)
32. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The V*-diagram: a query-dependent approach to moving knn queries. In: VLDB (2008)
33. Okabe, A., Boots, B., Sugihara, K., Chiu, S.: Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. Wiley, New York (2000)
34. Oppenheim, N.: Urban Travel Demand Modeling: From Individual Choices to General Equilibrium. Wiley, New York (1995)
35. Pedersen, J.: On the stability of crystal lattices. ix. Covariant theory of lattice deformations and the stability of some hexagonal lattices. In: Proceedings of the Cambridge Philosophical Society vol. 38 (1942)
36. Pei, J., Jiang, B., Lin, X., Yuan, Y.: Probabilistic skylines on uncertain data. In: VLDB (2007)
37. Sember, J., Evans, W.: Guaranteed voronoi diagrams of uncertain sites. In: CCCG (2008)
38. Sharifzadeh, M., Shahabi, C.: Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. In: PVLDB (2010)
39. Sistla, P.A., Wolfson, O., Chamberlain, S., Dao, S.: Querying the uncertain position of moving objects. In: Temporal Databases Research and Practice (1998)
40. Stallings, W.: Wireless Communications and Networks, 2nd edn. Prentice-Hall Inc, Upper Saddle River, NJ (2004)
41. Wang, P., Gonzalez, M.C., Hidalgo, C.A., Barabasi, A.-L.: Understanding the spreading patterns of mobile phone viruses. Sci. Exp. **324**(5930), 1071–1076 (2009)
42. Wong, R.C.-W., Özsu, M.T., Yu, P.S., Fu, A.W.-C., Liu, L.: Efficient method for maximizing bichromatic reverse nearest neighbor. PVLDB (2009)
43. Xu, J., Zheng, B.: Energy efficient index for querying location-dependent data in mobile broadcast environments. In: ICDE (2003)
44. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD (2003)
45. Zheng, B., Xu, J., Lee, W.-C., Lee, L.: Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services. VLDB J. **15**(1), 21–39 (2006)